MASTER THESIS

Analyzing Fileless Malware for the .NET Framework through CLR Profiling

Tom Leemreize

Faculty of Electrical Engineering, Mathematics and Computer Science

June 18th, 2021

UNIVERSITY OF TWENTE.

***Abstract*—Fileless malware is a currently ongoing threat, with high success rates at bypassing detection methods and infecting machines. Anti-malware solutions are continuously improving to tackle this threat by introducing new detection mechanisms. One of these mechanisms is the Antimalware Scan Interface, better known as AMSI, which has been a significant improvement to security in the .NET world. A new fileless malware technique based on the Dynamic Language Runtime is however able to bypass these new mechanisms, including AMSI. Therefore, a new method to tackle this threat is required. As a response, we propose a new method for analyzing fileless malware for the .NET Framework based on CLR profiling. As our method builds on top of the .NET profiling API, it is applicable to any application written for the .NET Framework. Our method has been successfully applied to current state-of-the-art malware samples to both analyze the samples and create signatures for their techniques. This in turn allows us to detect the usage of these techniques in new, unknown samples. From our analysis we discovered four distinct types of fileless malware techniques that are currently being used in the wild. These four types of techniques are reflection-based techniques, techniques statically invoking unmanaged code, techniques dynamically invoking unmanaged code, and techniques utilizing an embedded interpreter. Additionally, we also provide insights into the behaviour of these techniques by comparing their characteristics in more detail.**

***Index Terms*—Fileless malware, .NET Framework, Dynamic analysis, CLR Profiling, Dynamic Language Runtime**

I. Introduction

Malware, especially ransomware, is a currently ongoing threat. Ransomware is a special form of malware, where criminals encrypt files with a key. The target of the attack is then blackmailed by the criminals to pay money in exchange for this key. If the target does not pay the ransom, they will be unable to decrypt the files. Ransomware is just one type of malware, but many other types exist. Examples of malware types are banking Trojans, keyloggers, and spyware, to name a few.

Luckily, anti-malware solutions are usually able to tackle this issue. Anti-malware solutions such as ESET NOD32 [21], AVG [6], and Avira [7] can often detect malware before it can be executed. As a response to this, malware developers have started using new techniques as well. One of these techniques is fileless malware. With fileless malware, the actual malicious payload is never stored on the disk of the target. Because of this, anti-malware solutions cannot scan the malicious payload on disk and have to resort to other more complicated methods, such as scanning the memory or scanning network traffic. An example of fileless malware would be malware that downloads a malicious payload from a remote server, and executes it without writing the payload to disk. As fileless malware is able to bypass anti-malware solutions, it is a very effective technique for infecting machines. In a recent report, the Ponemon Institute has shown that fileless malware attacks are almost ten times more likely to succeed than traditional malware attacks [41]. Furthermore, a report by Malwarebytes Labs also predicts that the future of malware will most likely be fileless [26]. These reports show that fileless malware is indeed a relevant threat, and should be tackled sooner rather than later. Of course, anti-malware vendors are also continuously improving to tackle emerging threats, such as fileless malware. This cat-and-mouse game between malware developers and anti-malware vendors will most likely never end, as both parties keep evolving.

Recently, a new technique for fileless malware has popped up which is able to circumvent the current state-of-the-art detection methods [45]. When we refer to techniques for fileless malware, we are to referring to the approach used to achieve fileless behaviour. For the purposes of this research, this is limited to standard .NET Framework API calls and how these calls are utilized. An example of this would be a fileless malware technique based on reflection, which uses the .NET reflection API to achieve fileless behaviour. In Figure 1, an implementation of this technique can be seen. The implementation uses functions from the `System.Reflection` namespace, which contains the .NET reflection API.

```
/// <summary>
/// Loads a specified .NET assembly byte array and
    executes the EntryPoint.
/// </summary>
/// <param name="AssemblyBytes">The .NET assembly byte
    array.</param>
/// <param name="Args">The arguments to pass to the
    assembly's EntryPoint.</param>
public static void AssemblyExecute(byte[] AssemblyBytes
    , Object[] Args = null)
{
    if (Args == null)
    {
        Args = new Object[] { new string[] { } };
    }
    System.Reflection.Assembly assembly = System.
        Reflection.Assembly.Load(AssemblyBytes);
    assembly.EntryPoint.Invoke(null, Args);
}
```

Fig. 1. Function in SharpSploit [14] to load and execute a .NET assembly in memory using reflection (static functions inlined for clarity)

The new technique, dubbed BYOI for *Bring Your Own Interpreter*, uses a new feature added to the .NET Framework in version 4.5. This feature, called the Dynamic Language Runtime, allows scripting languages to be brought to the .NET Framework. An example of this is the IronPython project [2], which brings the Python programming language to the .NET Framework. By using this feature, a malicious payload can be written in Python, while it is executed by a .NET Framework application. The current detection measures fail to detect the payload as malicious, as they have been made to detect malicious .NET applications, and not Python scripts. One of these detection methods is the Antimalware Scan Interface, or AMSI, which Microsoft added to the .NET Framework in version 4.8 [38]. AMSI allows applications to interface with anti-malware solutions. This means that an application implementing AMSI can submit data to be scanned by an anti-malware solution. As this is implemented in functions of the .NET Framework API, it is not necessary for developers to interface with AMSI themselves in .NET Framework applications. By adding AMSI to the .NET Framework, Microsoft attempted to tackle the fileless malware threat for the .NET Framework, as it is commonly utilized with this goal in mind.

The new technique utilizing the Dynamic Language Runtime is however able to bypass AMSI. This is the case as the AMSI integration in the .NET Framework is made to scan entire .NET assemblies loaded through the reflection API, and not interpreted scripts. Additionally, as the scripts are interpreted, malware could decrypt and execute malicious payloads line by line. This way, the decrypted payload in its entirety is never present in memory, which can make detection even more difficult. The individual lines could be scanned before they are executed by the .NET runtime, however, the lines by themselves will most likely not indicate any malicious behaviour.

As we can see from previous reports and this new technique, current solutions are not sufficient to tackle the fileless malware threat, specifically for the .NET Framework. Therefore, we propose a new method for analyzing .NET Framework applications. The proposed method focuses on identifying the techniques used in a .NET Framework application. Current methods, such as the anti-malware solutions mentioned earlier, are targeted at specific samples. These methods detect malware by checking whether the hash of the executable is the same as a known malware sample. The method we propose in this paper is a more generic approach, which allows techniques used in fileless malware to be identified, such as the BYOI technique. Additionally, there is currently no analysis method that is capable of analyzing all .NET Framework application formats, to the best of our knowledge. In Section III, we explain the research questions this research aims to answer, and the goals of our approach in more detail.

Our proposed analysis method is based on the profiling API for .NET Framework applications. This in turn allows our method to be applied to any .NET Framework application, including PowerShell scripts. By utilizing the profiling API, we are able to create a call tree of all the function calls made by the application under analysis. As this call tree captures all of the function calls made by the application, it captures the entire behaviour of an application. By building signatures on top of the output of the call tree, we can automatically identify behaviour present in .NET Framework applications. This also includes the presence of fileless malware techniques, which are the primary targets of this research. As a result, this research provides signatures for currently used fileless malware techniques, allowing them to be identified in the wild. In Section IV, we show our analysis approach and the creation of the signatures in more detail. The implementation of our analysis method based on the profiling API is given in Section V.

By applying our tooling on several malware samples, including state-of-the-art frameworks that have been found used in the wild, we were able to identify several fileless malware techniques for the .NET Framework. We classified the identified techniques found into four categories, based on their behaviour:

- Reflection-based techniques;
- Techniques statically invoking unmanaged code (P/Invoke);
- Techniques dynamically invoking unmanaged code (D/Invoke);
- Techniques utilizing an embedded interpreter.

Each of these categories captures distinct behaviour found in the samples we analyzed. In Section VI, we compare these categories in more detail, and we show their unique characteristics.

In summary, in this research we both propose a completely new analysis method for .NET Framework applications and apply this new method to tackle the threat of fileless malware. We successfully used our method to analyze five different post-exploitation frameworks, covering samples written in two different .NET programming languages. Furthermore, we also showed the applicability of our method to other .NET Framework languages by analyzing an application written in the C#, F#, and Visual Basic programming languages. Our analysis method was able to achieve the goals and requirements we had set, and is a valuable addition to malware research for the .NET Framework. Additionally, we showed that our method was successfully able to identify fileless malware techniques in samples from current state-of-the-art malware frameworks. As a result, we also present a list of the fileless malware techniques we encountered being used in the samples in our dataset. We then compared these techniques in more detail to find out exactly how they differ from each other, and what the advantages and disadvantages of these techniques are. This information will in turn be used to improve current detection methods for the .NET Framework.

II. Background

In this section, we explain the necessary background information on which the research is built. First, we give a short introduction of the .NET Framework and its components, after which we cover the Dynamic Language Runtime in more detail. Next, we discuss PowerShell and the features that make it relevant in the context of fileless malware. Lastly, we cover fileless malware itself and the currently available detection techniques for fileless malware for the .NET Framework.

A. .NET Framework

The .NET Framework was initially released by Microsoft in June 2000 as a competitor to the Java 2 Enterprise Edition (J2EE) of Sun Microsystems [23]. The .NET Framework provides programmers with a wide library of features, allowing developers to significantly speed up their development. Similar to the Java Virtual Machine, the .NET Framework is platform agnostic. Additionally, the .NET Framework is included by default on Windows systems since Windows XP Service Pack 2 [43]. The .NET Framework consists of two primary components, namely the Common Language Runtime (CLR) and the Framework Class Library. The Framework Class Library provides programmers with a set of functions to be used in their own codebases [30]. An overview of the architecture can be found in Figure 2.

The CLR is at the core of the framework, and functions as a layer between the Common Intermediate Language (CIL), to

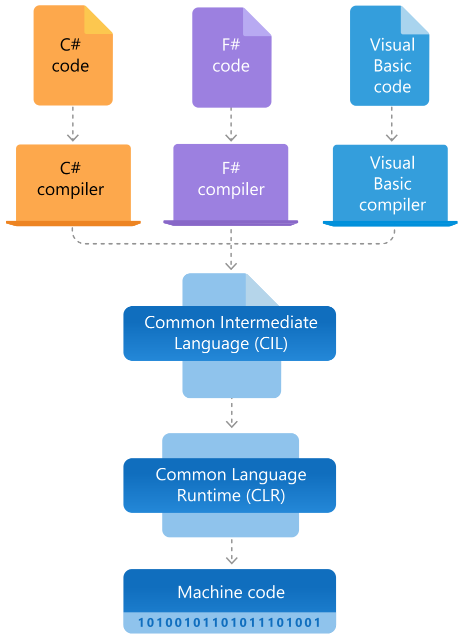


Fig. 2. Overview of the .NET Framework architecture [36]

which languages targeting the .NET Framework are compiled, and native machine instructions. The CLR can be compared to the Java runtime, which provides similar functionality for Java bytecode. The runtime is what allows programmers to use any programming language targeting the .NET Framework, while still being able to take full advantage of the features of the CLR, as the runtime is language-neutral and utilizes the CIL [23].

These components integrate together as follows. When a developer writes an application in a language that supports the .NET Framework, such as C#, the code is compiled to the CIL. This compiled code is then stored in an assembly format, such as *.dll* or *.exe* [36]. When these assemblies are executed, the CLR takes the CIL instructions and compiles them to native machines instructions. This ensures that the assembly can run on any system for which there is a runtime. Unlike the Java Virtual Machine, the CLR does not function as an interpreter. The CIL code is always just-in-time compiled to native instructions [47].

B. Dynamic Language Runtime

The Dynamic Language Runtime (DLR) builds on-top of the Common Language Runtime and provides support for dynamic languages to the .NET Framework [28]. The most notable usage of the DLR is the IronPython project, which brings the Python programming language to the .NET Framework [2]. Aside from enabling dynamic languages to be ported to the .NET Framework, the DLR also allows the dynamic languages to make use of all the features and libraries of the .NET Framework. This means that a scripting language, like Python, can be used to access the entire .NET API. Another added benefit is that the dynamic features are also accessible in C# with the use of the `dynamic` keyword.

The DLR enables tasks that would normally be done at compile time to be executed at runtime. An example of this are dynamic types. Dynamic types are types which are not yet implemented at compile time, but will get implemented during runtime. This can be useful when parsing values from external sources, such as an HTML document. Instead of having to use functions such as `html.getProperty("body")` to access the body of an HTML document, it would be possible to simply implement this dynamically and access it using `html.body` [28]. This can make code much simpler to comprehend and use, especially in long chains involving multiple get and set operations. This dynamic behaviour also applies to methods, allowing methods to be implemented during runtime. As these methods are not present in the executable itself, the DLR can therefore allow developers to execute code in memory, without having to recompile. While this can speed up development significantly, it also creates a new surface for potential abuse by nefarious actors. This abuse is explored in Section II-E.

C. PowerShell

PowerShell is a command-line shell and scripting language originally released by Microsoft with the purpose of task automation and configuration management [37]. With this purpose in mind, PowerShell is generally not blacklisted on systems as it is used by system administrators for configuration and automation. Furthermore, PowerShell is also pre-installed on all Windows systems since Windows 7 and Windows Server 2008 R2 [25]. PowerShell is built on top of the .NET Framework and therefore also allows full integration with assemblies written for the .NET Framework, in addition to non-.NET PE binaries (such as *.exe* and *.dll* files). These two properties make PowerShell an interesting tool for attackers, as it is both powerful and widely available.

Additionally, PowerShell scripts allow users to execute entire scripts and executables completely in memory, without having to drop a file on the hard drive of a system. While useful for benign users to allow quick execution of scripts, malicious users can abuse this feature to reduce the amount of traces left behind. This bypasses file-based anti-malware techniques, as there are no files that can be analyzed since everything is done in memory [25].

D. .NET Malware

The introduction of PowerShell and the .NET Framework has made life easier for benign developers, however nefarious developers took note and have also started using these tools. Research performed by Kaspersky Labs has shown that unique .NET malware detections have grown by 1600% between 2009 and 2015 [42]. In 2009 the detections were less than a million, yet in 2015 the number of detections is closer to 15 million. This growth confirms that nefarious malware developers have been catching on onto the usefulness of the .NET Framework as well. Additionally, in 2018, 2019 and 2020 the Emotet trojan has been one of the top five threats for Windows [26]. Emotet utilizes PowerShell to download and execute other malicious payloads [17], [48], showing the effectiveness of utilizing the .NET Framework.

The malicious uses of the .NET Framework come in many forms, ranging from ransomware to spy campaigns of governments [43]. Pontiroli and Martinez [43] took several kinds of .NET malware and showed their workings. In their analysis it can be seen that .NET malware can achieve the same feats as classical platform specific malware. CoinVault, a .NET ransomware variant, utilizes the functions included with the Framework Class Library to encrypt the files of a target with relative ease. Functions to determine whether the malware was ran within a virtualized environment can also be found. Additionally, the malware also uses a technique dubbed as *RunPE*. With *RunPE*, the payload of the malware is loaded by a legitimate process in memory. This means that the malicious payload never touches the disk. Malware using these properties are so called "fileless malware", and will be described in more detail in the next section.

Similar things can be achieved with PowerShell scripts, as PowerShell also fully integrates with the .NET Framework. PowerShell also allows the malicious payloads to be encoded, which defeats many of the built-in security features by Microsoft [43]. Similar to *RunPE*, the real power of PowerShell however lies in its capabilities to execute malware in memory. The next section will cover this in more detail and give examples of famous PowerShell frameworks.

E. Fileless Malware

The dynamic nature of PowerShell, which allows the in-memory execution of scripts and assemblies, has led to a new era of malware, namely the one of fileless malware. As the name implies, fileless malware is a variant of malware where everything is done in memory and the malicious payload never "touches" the hard drive.

According to Baldin [8], fileless malware can typically be split up into four categories: documents, scripts, code in memory, and "living off the land"-techniques. Fileless attacks using documents embed malicious scripts, such as a macro in the popular Microsoft Word software, in order to infect the machine. It should be noted that this variant is not truly fileless, as the document is present on the system. It is, however, still considered fileless due to the use of embedded scripts. The next category is that of scripts, which are typically executed through a browser. This variant can however also not be considered truly fileless, as modern browsers cache downloaded scripts on the disk in order to lower load times for users.

The next two categories are more interesting from the perspective of an attacker as the malicious payload only exists in memory. The first of these two categories uses another wrapper executable to execute the malicious payload in memory. The malicious payload can be embedded in the wrapper executable as encrypted shellcode, or downloaded from a remote server. In this scenario the wrapper loads the payload into memory and executes it, therefore never dropping the actual malicious payload on disk. This makes malware detection much harder as there is no malicious file of which the signature can be checked for a known malware match. The last category, "living off the land"-techniques, are the most interesting when considering the .NET Framework. This type of fileless malware utilizes legitimate processes for malicious purposes. This is similar to the last category, except the wrapper executable is a legitimate process instead. PowerShell is a commonly used target for these attacks, for the reasons mentioned in Section II-C: it is rarely blacklisted and pre-installed on most systems. Additionally, PowerShell will be less likely to raise suspicion when seen running on a system compared to an unknown executable. This makes these types of malware much harder to detect, as the processes running the code appear to be legitimate.

An example of a framework using the "living off the land"-technique for PowerShell is PowerSploit [5]. PowerSploit contains several scripts, of which some could be considered fileless malware. An example of such a script is the *Invoke-ReflectivePEInjection* script, which enables an attacker to load and execute a PE binary reflectively, which means that the binary is executed without having been written to the disk, from the PowerShell process.

Other projects, such as PSAttack [22] take the "living off the land"-approach a step further. PSAttack combines several modules, one of which being PowerSploit, and builds on top of it. PSAttack uses obfuscated and encrypted payloads, which are decrypted in-memory. Therefore, aside from being a legitimate process, the decrypted malicious payloads also never touch the disk. This makes it significantly harder for anti-malware programs to detect the payloads, as these programs would not be able to detect whether the payloads are malicious in the first place.

As mentioned earlier, the Dynamic Language Runtime has also been a target for nefarious actors. These actors have realized the potential of the DLR for fileless malware. An example of the abuse of the DLR is the SILENTTRINITY project [12]. The SILENTTRINITY project is a post-exploitation framework which uses the DLR to allow attackers to use scripting languages to execute tasks, as opposed to having to compile C# code to achieve the same results. This gives attackers much more flexibility when writing their payload. Another major advantage of this is that by embedding the interpreter of the scripting language into the initial C# payload, AMSI only scans the interpreter and not the malicious script. This happens because AMSI only scans assemblies loaded through reflection, which is not the case when interpreting the script. In short, this technique allows for an entirely new format of fileless malware for the .NET Framework.

F. Detection Techniques

As fileless malware, and malware in general, is not new in the world of cyber-security, detection techniques to detect these kinds of attacks have already been developed for the .NET Framework. Microsoft has added their own features to Windows to protect against the misuse of PowerShell, and has also added extra logging features to aid in the detection of these attacks.

1) Antimalware Scan Interface (AMSI): One of the most significant additions by Microsoft has been the addition of the Antimalware Scan Interface, more commonly known as AMSI. AMSI allows any application to interface with an anti-malware product. This means that any application can submit files, memory, streams, URLs, IPs and more to be checked for malicious contents [39]. Additionally, any anti-malware vendor can integrate with AMSI to provide a scanning engine for malware.

AMSI is tightly integrated into several components of Windows itself, with the most important being PowerShell, Windows Script Host, VBScript and Office macros. For this research the integration with PowerShell is the most interesting of these components, as it attempts to tackle the fileless malware problem. Microsoft has also added AMSI scanning for all loaded assemblies in the .NET Framework in .NET version 4.8 [38]. This includes assemblies loaded from memory, indicating that Microsoft is also actively tackling fileless malware on this front.

As mentioned earlier, PowerShell scripts can be encrypted and obfuscated. By doing this, malware developers can avoid typical static anti-malware solutions which compare signatures or the contents of files to known malicious samples. AMSI can however tackle this issue by analyzing the samples after they have been deobfuscated and decrypted, but right before they are compiled and executed [44]. As AMSI integrates with Windows Defender by default, it can provide protection against fileless malware attacks out of the box.

By using the DLR it is however possible to bypass AMSI [11]. This is exactly what SILENTTRINITY does by embedding an interpreter, as was mentioned in the previous section. This shows that while AMSI is step in the right direction, it is still not sufficient to effectively detect fileless malware attacks.

2) Event Tracing for Windows (ETW): Another addition made by Microsoft is Event Tracing for Windows, a tracing facility that enables applications to log events [27]. These events can then be monitored in real-time or be written to a log file. Windows supplies many event providers by default, but the most relevant in the context of fileless malware is the built-in logging to PowerShell.

Since PowerShell version 5, all code executed within PowerShell can be logged [35]. This applies to any application using the PowerShell engine, so not just the PowerShell shell [4]. This provides additional ways of analyzing PowerShell scripts for malicious contents, which can be used in detection techniques.

It should be noted that PowerShell itself also has a means of emitting events when suspicious scripts are executed. These events could also aid in the detection of fileless malware, as they indicate when a script executes functions commonly associated with fileless malware. These suspicious functions include those used for dynamic assembly building, Win32 API calls, and PowerShell obfuscation techniques[1]. These events could therefore also be used to detect fileless malware for PowerShell.

The major downside of the script block logging PowerShell provides is that obfuscation of the script persists to the logs [10]. Collberg and Thomborson define obfuscation as a way to achieve security through obscurity [15]. They describe obfuscation as the transformation of a program into another program, where the obscurity is maximized. The behaviour of the program should be preserved during these transformations. Obfuscation can make it incredibly difficult to figure out the functionality of a script. Furthermore, removing layers of obfuscation is extremely time consuming when done manually. Programmatically removing obfuscation is also infeasible, as there are simply too many ways in PowerShell to achieve the same functionality. Methods used for obfuscation can be combined as well, resulting in even more obfuscation layers that have to be removed. This detection technique is therefore not sufficient to detect malicious PowerShell scripts, as obfuscation is a common occurrence in PowerShell.

While the detection techniques added to Windows can help defend against most forms of malware, they are not foolproof. Fileless malware utilizing the DLR is undetected by any of the detection techniques mentioned. For this reason, we developed a new method capable of analyzing and potentially detecting this variant. In Section V, we describe our method, using the profiling API of the CLR, in more detail.

III. Motivation

In this section, we explain the motivation behind this research and the goals we want to achieve with this research. First, we explain the problems this research addresses and the relevancy of these problems. Then, we list the research questions formulated to solve the mentioned problems. Lastly, we discuss the goals and requirements that the answers to the research questions should adhere to.

A. Problem Statement

As we showed in Section I and Section II-E, the on-going cat-and-mouse game between malware developers and anti-malware vendors has led to malware becoming more and more sophisticated. One of the results of this arms race has been the development of fileless malware. Fileless malware can be an incredibly powerful tool when trying to circumvent detection by anti-malware vendors, as there are no malicious files that can be scanned.

In a report by the Ponemon Institute [41], they show that fileless attacks are on the rise. The reports shows that 29% of the attacks in 2017 were fileless, which is an increase of almost 50% compared to the year before. Additionally, the report also claims that fileless attacks are almost ten times more likely to succeed than attacks utilizing files. In fact, their survey shows

[1]https://github.com/PowerShell/PowerShell/blob/79f21b41de0de9b2f68a19ba1fdef0b98f3fb1cb/src/System.Management.Automation/engine/runtime/CompiledScriptBlock.cs#L1546-L1829

that 42% of the companies surveyed have had their data or infrastructure compromised due to fileless malware attacks.

In another report by Malwarebytes Labs, they too state that the future of malware will most likely be fileless [24]. The report by Malwarebytes Labs also mentions that PowerShell has been used in successful attacks in the past years, such as the infamous Emotet trojan. Additionally, in the report they mention that the traditional approach of only analyzing files on disk is simply not sufficient anymore.

As we can see from these reports, fileless malware attacks are getting increasingly more popular due to their success rates, especially the attacks based on PowerShell. Therefore, the primary goal of this research is to study the current fileless malware techniques for the .NET Framework, which includes PowerShell. In this research, we consider a fileless malware technique to be the approach used to achieve fileless behaviour. An example of this would be a fileless malware technique based on reflection, which uses the .NET reflection API to achieve fileless behaviour. Literature review has pointed out that current research studying fileless malware for the .NET Framework is mostly targeted at PowerShell scripts. However, the .NET Framework is much broader than just PowerShell scripts, meaning there are many more attack vectors to be studied. With this research, we want to extend current research by studying fileless malware across the entire .NET Framework. Furthermore, we specifically target a new technique for fileless malware for the .NET Framework, based on the Dynamic Language Runtime. Utilization of the Dynamic Language Runtime for malicious purposes is a fairly new development and is currently still unresearched. The result of this research will therefore be a more complete picture of the currently available attack vectors for fileless malware for the .NET Framework.

In order to achieve this, we analyze fileless malware variants across the entire .NET Framework. Malware for the .NET Framework can however come in many forms, due to the different formats that can run on the Common Language Runtime. As we mentioned, PowerShell is one example of this, but regular .NET executable files also run on top of the Common Language Runtime. To the best of our knowledge, there currently is no method that makes it possible to analyze samples across the entire spectrum of the .NET Framework. In other words, there is no solution that can both analyze PowerShell scripts and regular Windows .NET executables. Therefore, a method that makes it possible to analyze these formats is required to study fileless malware for the .NET Framework. Aside from analyzing samples and creating an overview of the current techniques, we also want to know how these techniques differ. We therefore need to analyze the behaviour of these samples to create a list of characteristics for the different fileless malware techniques. This list of characteristics will allow for the identification of the techniques we discovered, and allow us to compare the techniques. This comparison can then show us the similarities and differences between the discovered techniques.

In short, this research aims to both create an overview and analyze the current fileless malware techniques for the .NET Framework. Special attention will be paid to the fileless malware technique based on the Dynamic Language Runtime, as this variant is able to bypass current state-of-the-art detection methods, as we mentioned in Section II-F. The result of the analysis process will be a list of characteristics that allow for the identification of the different fileless malware techniques. Additionally, these characteristics also show how the techniques differ from each other. This information can then be used to create a better understanding of the current threats and allow for the development of better detection mechanisms.

B. Research Questions

The primary objective of this research is to study the current fileless malware techniques for the .NET Framework. Additionally, we pay special attention to the new fileless malware technique based on the Dynamic Language Runtime, as it is able to bypass current state-of-the-art detection methods such as AMSI.

We formulated several research questions in order to achieve these objectives. The answers to these research questions will lead to gaining a better understanding of the workings of fileless malware for the .NET Framework. The research questions this research aims to answer are as follows:

1) How can different .NET Framework applications be compared, regardless of their programming language?
2) What are the current fileless malware techniques for the .NET Framework?
3) What are the characteristics of fileless malware techniques using the Dynamic Language Runtime?
    a) What are the differences and similarities between fileless malware techniques using the Dynamic Language Runtime and other kinds of fileless malware techniques for the .NET Framework?

We formulated these research questions in order to gain insight into fileless malware for the .NET Framework.

The goal of the first research question is to find a method that allows different .NET Framework applications to be compared. Applications written for the .NET Framework can come in many forms, such as PowerShell scripts and executables written in C#. As the research is focused on fileless malware for the .NET Framework, it is required to be able to compare these formats. The result of having a method that can do this will be a common ground for comparing .NET Framework applications. This in turn allows us to analyze the applications and create signatures that can be applied to any .NET Framework application. These signatures will be required to identify the used techniques and to answer the other research questions.

The second research question is aimed at uncovering the current and researched techniques for fileless malware for the .NET Framework. This will indicate the current state of the fileless malware landscape for the .NET Framework. Additionally, this will result in other fileless malware techniques that we can compare to fileless malware techniques utilizing the DLR. Together with the next research question, this would then allow for us to show the similarities and differences between this new technique and other techniques. This will in turn give researchers and security specialists a clear image of how the new technique differs from existing techniques and how it could be detected.

The next research question, together with its sub-question, should point out unique characteristics of malware using the new technique. The characteristics of the malware are features that allow it to be distinguished from other pieces of malware and benign software. These characteristics can for example be found through the analysis of ETW traces, or .NET API calls made by the malware. After this information is known for fileless malware techniques for the DLR, the differences and similarities between types of fileless malware for the .NET Framework can be analyzed. We formulated this research question to analyze these differences and similarities.

In summary, the answers to these research questions will give a good indicator on the current state of the fileless malware landscape for the .NET Framework. In addition to showing what the current fileless malware techniques are, we also study their characteristics by analyzing them in more detail. This will in turn show whether the current techniques are comparable to the new technique utilizing the DLR, therefore showing the risks of the new forms of fileless malware as well. In Section IV, we explain the approach to answering these research questions in more detail.

C. Goals & Requirements

In order to compare .NET Framework applications, a method that captures the behaviour of a .NET Framework application is required. The output format of this method should be the same, regardless of the application it was applied to. For the purposes of this research, this means that for both PowerShell scripts and .NET Framework executables the output format of this method should be the same. Additionally, the output of the method should also be deterministic whenever possible. This means that when the method is applied to an executable multiple times, the output should be the same every time. In cases where the executable utilizes randomness to call functions, it is however not possible for the output to be fully deterministic. This should however not influence the results of this research, as the core calls that make the fileless functionality possible should still be present in the output. The method should also not rely on the availability of the source code of the application, as this is often not available for malware.

Another major requirement is that it should be possible to parse the output programmatically. In practice this means that the output should be either text based, or a structured binary format. An image displaying the behaviour of the application would therefore not be sufficient, as it would be non-trivial to parse programmatically. A structured format will also allow YARA rules to be written to the identify malicious techniques used. YARA rules allow researchers to specify a description of malware, which can then be used for automated classification [51]. With this, a signature could be written for a specific characteristic of a technique. This signature could then be applied across all .NET Framework applications, including those based on the DLR. This would then quickly show whether two applications are utilising similar techniques. Additionally, as the YARA rules capture the characteristics of the techniques, the rules can be compared to show differences between the techniques.

IV. Approach

In this section, we explain our approach to answer the aforementioned research questions. The approach is split into four phases: profiling, analysis, signature creation, and comparison. The profiling phase takes a malware sample as input, while the other phases take the output of the previous phase as input. The output of the final phase is a list of techniques that share similar characteristics and techniques that do not, including those characteristics. A global overview diagram of the approach can be found in Figure 3.

In the first phase, profiling, we collect a list of all the calls performed by the malware sample in the form of a tree. In this call tree, each node corresponds to a function call. Each branch from parent to child indicates a function call from the parent function to the child function. When combined, these calls provide insights into the behaviour of the malware sample. This call tree provides the basis on which we perform the analysis and from which we create signatures. Next, we continue to the analysis phase. We analyze the call tree to find the calls that the sample uses to load and execute the payloads in memory. The combination of these calls are what we define as the fileless malware technique. By analyzing source code, when available, or reverse engineering the malicious sample, we can isolate calls belonging to specific techniques. These calls are then used in the next phase, which is the signature creation process. As the scope has been reduced significantly, we can now analyze the calls belonging to fileless malware techniques in more detail. For each technique, we first identify the calls that are absolutely necessary. After this, we make signatures for those techniques using the absolutely necessary calls. As a result, these signatures are now able to effectively identify these techniques. By limiting the signatures to necessary calls specific to the techniques, we can increase the accuracy of our signature. The very last step of this process is to compare the signatures for the different techniques.

The result of this process allows us to achieve our goals of showing the currently available fileless malware techniques and pointing out the differences and similarities between them. The remainder of this section explains in detail how the steps for each phase are executed.

A. Profiling

The first step in understanding how a .NET Framework malware sample functions, is to profile the malware sample. By running the sample under the profiler, we make a high-level overview of the functionality of the sample. The sample is

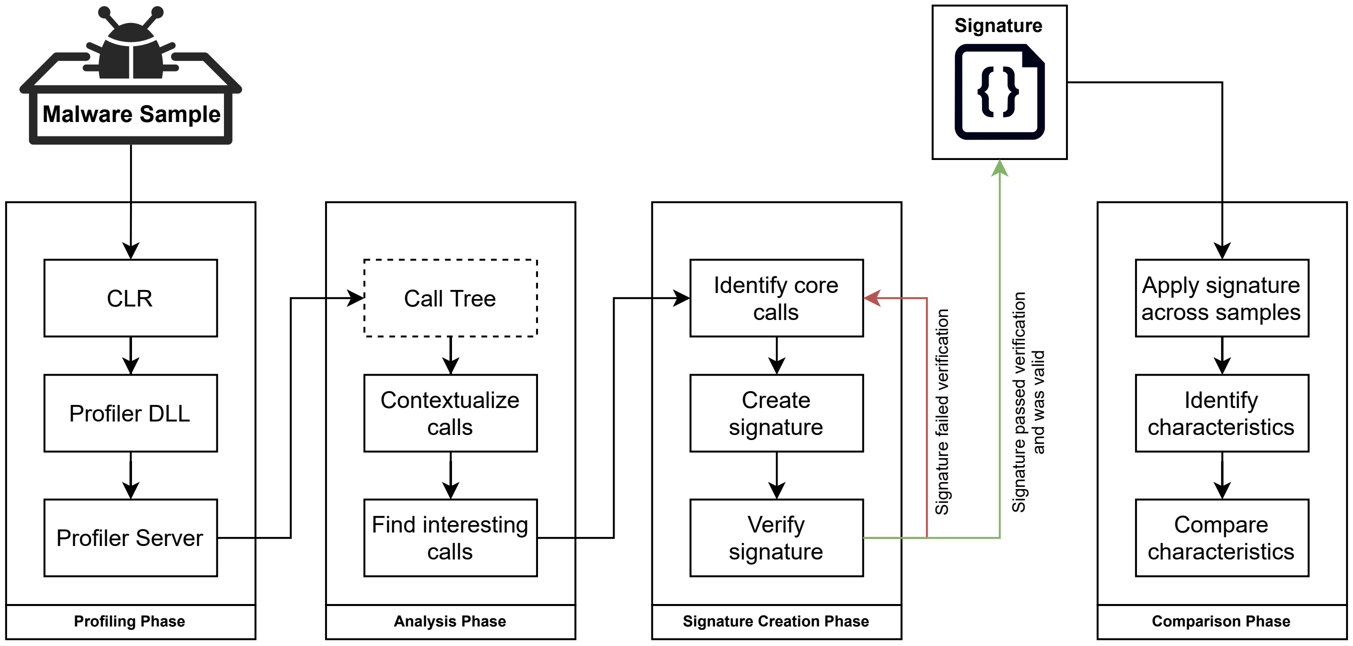


Fig. 3. Overview of our malware analysis approach

profiled with the profiler described in Section V to gain insight into the functionality of the sample. The result of running the sample under the profiler is that every call made by the sample is logged. This list of calls is then used for further analysis.

Additionally, code obfuscation techniques will also not affect analysis this way. Obfuscation can make the application harder to reverse engineer, but the functionality will remain the same. As functionality remains the same, the .NET calls to achieve the functionality still have to be present in the application. Therefore, calls hidden through obfuscation are also included in the call log created by the profiler.

To profile samples, we first need to configure the CLR to connect to the profiler DLL when the CLR starts. This is done by setting the appropriate environment variables: `COR_ENABLE_PROFILING`, `COR_PROFILER_PATH`, and `COR_PROFILER`. The first variable indicates that a profiler should be used, while the last two allow the CLR to locate the profiler files. The result of this is that the profiler is attached to any .NET process that is started. This includes any other .NET process started by the sample that is being analyzed. It is possible for an application to detect the presence of a profiler by checking the environment variable, and adjust its behaviour. In case this happens, we should be able to see this process in the call log, and either circumvent it by modifying the sample, or decide to exclude the sample. As this is a new approach to analyzing malware, however, we do not expect to see these checks in currently available malware samples. It should be mentioned that the profiler DLL does not process the calls, but sends them to a separate server component instead. This server component keeps track of all the calls made by the sample, and writes them to a file. Splitting the profiler into separate parts is done for performance purposes, and described in Section V in more detail.

Now that the profiler has been configured successfully, we execute the malicious sample. As the CLR executing the sample is instructed to use the profiler, all calls made by the sample are sent to the profiler. In turn, the profiler logs all function calls made by the sample and stores these in a tree-like structure. This structure was chosen as it allows us to represent all functions, and the runtime calling relationships between these functions in a text format. In Section V we describe the structure of the call tree in more detail. An example of the tree-like structure can be seen in Figure 4. This figure shows Covenant [13], a post-exploitation framework, preparing to load Mimikatz into memory. Mimikatz is a tool that implements several methods to retrieve Windows credentials from a system, and is often included in post-exploitation frameworks [20].

This structure shows which calls lead to which other calls, providing more information about the control flow of the sample. Furthermore, the time taken for each function to execute is also stored. This information is then combined by the profiler, creating a complete picture of the execution of the sample. Whenever the sample connects to the profiler, the lifetime of the profiled process is tracked. When all profiled processes are terminated, the profiler stops logging and stores the saved calls. The logs can now be analyzed, allowing for the comparison between different samples.

The profiler provides a common ground for .NET malware comparison, independent of techniques used. This means that the profiling step is the same for every .NET Framework application, including PowerShell scripts. The next step is to analyze the collected call log for malicious techniques.

```
0.00% 0.349ms 1 calls Task.Execute
    0.00% 0.349ms 1 calls SharpSploit.Credentials.Mimikatz..cctor
    0.00% 0.349ms 1 calls SharpSploit.Credentials.Mimikatz.Command
        0.00% 0.349ms 1 calls System.Reflection.Assembly.GetExecutingAssembly
        0.00% 0.349ms 1 calls System.Reflection.RuntimeAssembly.GetManifestResourceNames
        0.00% 0.349ms 1 calls SharpSploit.Misc.Utilities..cctor
        0.00% 0.349ms 1 calls SharpSploit.Misc.Utilities.GetEmbeddedResourceBytes
        0.00% 0.349ms 1 calls SharpSploit.Misc.Utilities.ReadFully
        0.00% 0.349ms 1 calls Kernel32..cctor
        0.00% 0.349ms 1 calls SharpSploit.Execution.PE.Load
            0.00% 0.349ms 1 calls SharpSploit.Execution.PE..ctor
            0.00% 1.746ms 5 calls SharpSploit.Execution.PE.get_Is32BitHeader
            0.00% 8.032ms 23 calls System.Runtime.InteropServices.Marshal.PtrToStructure
            0.00% 0.349ms 1 calls System.Runtime.InteropServices.Marshal.SizeOf
            0.21% 933.093ms 2672 calls System.Runtime.InteropServices.Marshal.ReadInt16
            0.21% 929.252ms 2661 calls System.Runtime.InteropServices.Marshal.ReadInt64
            0.25% 1106.658ms 3169 calls System.Runtime.InteropServices.Marshal.WriteInt64
            0.04% 197.313ms 565 calls System.Runtime.InteropServices.Marshal.ReadInt32
            0.04% 187.185ms 536 calls System.Runtime.InteropServices.Marshal.PtrToStringAnsi
            0.04% 187.185ms 536 calls System.String.Equals
            0.04% 177.407ms 508 calls System.StubHelpers.CSTRMarshaler.ConvertToNative
            0.00% 0.349ms 1 calls System.Runtime.InteropServices.Marshal.GetDelegateForFunctionPointer
        0.00% 0.349ms 1 calls SharpSploit.Execution.PE.GetFunctionExport
        0.00% 0.349ms 1 calls System.Runtime.InteropServices.Marshal.GetDelegateForFunctionPointer
        0.00% 0.349ms 1 calls System.Runtime.InteropServices.Marshal.StringToHGlobalUni
        0.00% 0.349ms 1 calls System.Threading.Thread..ctor
        0.00% 0.349ms 1 calls System.Threading.Thread.SetStartHelper
        0.00% 0.349ms 1 calls System.Threading.Thread.Start
```

Fig. 4. Profiler output example of Covenant starting a Mimikatz task

B. Analysis

The result of the profiling phase is a tree-like structure of all calls made by the malware sample. In the analysis phase, we analyze the call tree and extract the calls that are relevant for the techniques used by the sample. For the purposes of this research, we are specifically interested in the calls related to fileless malware. These are the calls that load an assembly reflectively, or get a function pointer that is then executed. In order to find these calls, we first need to consider the functionality of the malware sample. By taking the functionality of the sample into account, we can find out in which context the calls in the tree are used. In samples where source code is available, we analyze the source code to give clues on the functionality. When source code is not available, knowledge of the malware sample can be used instead. For example, a malware sample with the intention of stealing passwords might contain functions with *"password"* in their names.

An example of this is the Covenant output shown in Figure 4, in which we can find several references to Mimikatz [20]. In order to create this output log, we started a Mimikatz task in Covenant, so searching for references to Mimikatz can give us an indication of where to start looking in the log. In this example, it indeed points us towards the section in which Mimikatz gets loaded into memory and executed. If we had not searched for Mimikatz and instead looked through the output manually, we would have to look through all of the 4000 lines of the output. Instead, by bringing the calls into context we are able to reduce the amount of calls to consider for signature creation significantly.

As the code for SharpSploit, the library used by Covenant to execute Mimikatz, is open source, we can compare our profiler output to the source code. In the call tree seen in Figure 4, we can see calls to `PE.Load`. The name of this function strongly indicates that it might contain the functionality for loading a

```
/// <summary>
/// Loads a PE with a specified byte array. (Requires Admin)
/// **(*Currently broken. Works for Mimikatz, but not arbitrary PEs*)
/// </summary>
/// <param name="PEBytes"></param>
/// <returns>PE</returns>
public static PE Load(byte[] PEBytes)
{
    PE pe = new PE(PEBytes);
    if (pe.Is32BitHeader)

    [..] // Code omitted for brevity

    // Call dllmain
    threadStart = IntPtrAdd(codebase, AddressOfEntryPoint);
    main dllmain = (main)Marshal.GetDelegateForFunctionPointer(threadStart, typeof(main));
    dllmain(codebase, 1, IntPtr.Zero);
    // Console.WriteLine("Thread Complete");
    return pe;
}
```

Fig. 5. Start and end of SharpSploit source code to load the Mimikatz executable [14]

PE into memory. After looking up the relevant function in the source code of SharpSploit, we can indeed see that this is the case. An excerpt of the function in SharpSploit is shown in Figure 5, and in this function we can find the same calls as were present in our profiler output. This example shows how we can use the functions present in our profiler output and the source code to find the relevant calls for a fileless malware technique.

Now that the relevant calls for a technique in the malware sample are known, we can create the signatures for the technique. This is the next step of the process, where we convert the calls into a signature capable of detecting the technique.

C. Signature Creation

The next phase is the signature creation phase. The previous phase provided a list of calls belonging to a specific technique used in the malware sample. During the signature creation phase, we analyze the list of calls to find the calls absolutely necessary to perform the technique. The absolutely necessary calls are the ones that cannot be easily removed or replaced to achieve the functionality of the technique. When these necessary function calls are not present or replaced, we consider it a different technique.

An example of this is when invoking a method using reflection, the `RuntimeMethodInfo.Invoke` function is called. This function is the lowest level call involved when calling a method using reflection. Therefore, this function call should always be present when calling functions using reflection. Another example can be found in the source code for SharpSploit, which can be seen in Figure 5. At the end of the function, a call to create a delegate using `GetDelegateForFunctionPointer` can be seen, after which the delegate is called. When a technique mixes unmanaged and managed code, this function is necessary to call an unmanaged function from managed code. As the technique used in SharpSploit does not involve a call to `RuntimeMethodInfo.Invoke`, we consider it a different technique from a technique that does use this call.

Identifying these calls requires an understanding of the .NET Framework API and the techniques used, and can differ per sample analyzed. As the calls are all present in the standard .NET Framework API, we can look up the official documentation for these functions. The documentation will then indicate whether these functions are relevant in the context of fileless malware. This can for example be a function that makes it possible to load or execute data in memory. If more calls are selected than necessary, a small change in the implementation of the technique would cause the signature not to match. On the other hand, a signature that matches a very small amount of calls might result in false positives. Therefore, it is important to carefully select the calls used in the signature.

After we have created the signature, we verify it on the sample it was created from. If the signature correctly matched on the sample, the rule has been written correctly without syntax errors. In order to verify that the rule was not too specific, the sample needs to be executed under the profiler again. This results in another call log for the same application. This new call log can however have some slight changes, due to randomness in the application or due to other options being chosen by the user. To verify the signature, we match it on this new call log as well. If the signature matches, the signature is correct and can be used to identify the technique. In case the signature does not match, the calls used to create it should be reconsidered. Additionally, in order to verify whether the signature does not lead to false positives, we match it on call logs from other samples as well. This should not result in matches unless the two samples utilize the same fileless malware technique. If the signature matches on other samples that utilize different techniques, the signature should be made more specific by adding additional calls.

After we have performed this process for all the samples and multiple signatures are available, we can compare the signatures. This comparison will show differences between the techniques, and is the next and final phase of the approach taken.

D. Comparison

One of the goals of this research is to uncover whether there are differences between techniques that use the DLR and those that do not. Applying the signatures will partly show the differences between techniques, as signatures for samples using similar techniques should theoretically match on both of the samples. In case they do not, we investigate what the differences are between the samples causing the signature not to match. In order to do this, we compare the signatures that were made during the previous phase in more detail. As was mentioned in the previous phase, the signatures are made to capture the essence of the technique. Therefore, they provide an excellent way to compare the techniques to each other.

The functionality of fileless malware techniques can be broken down into roughly two stages:

1) Loading the malware into memory
2) Executing the malware

In order the find similarities and differences between the different kinds of malware, both of these stages are compared separately. For similarities in the first stage, we check how the different samples load the malicious payload into memory. We then sort this into categories for each loading technique. The categories are based on the .NET Framework API calls used to implement the technique. For example, if two techniques use different reflection calls to load a payload into memory, we consider them both part of techniques that use “reflective based loading”. This process depends on the samples that were analyzed, and therefore there is no predefined set of categories. The sorting into categories is done to compensate for small differences in the implementation of different techniques. It might occur that two samples implement the same technique, but in a slightly different fashion, or using different calls that perform the same actions. Sorting into categories therefore prevents us from labeling each implementation of a technique as a distinct new technique.

The comparison process for the execution stage is similar. It should be noted that the technique used for loading malware into memory usually goes hand in hand with the technique used to execute it. A malware sample that uses unmanaged code to load the payload into memory, will most likely also use unmanaged code to execute it. Nevertheless, as this might not always be the case we perform the same steps for the execution stage as the loading stage.

These steps provide us with a list of techniques that share similar characteristics, and techniques that do not. This in turn also shows the difference in characteristics between the DLR-based techniques and other techniques.

After we have executed all of these phases, the information required to answer our research questions is available.

V. Implementation

As was mentioned in Section IV, we developed a profiler to log the function calls of .NET applications. Before developing the profiler, we considered several other methods capable of comparing .NET applications, each with their own advantages and disadvantages. In this section, we discuss the implementation of the profiler in more detail and briefly go over the alternatives we considered.

A. *GroboTrace*

The developed profiler builds on top of GroboTrace [1]. We initially used the commercially available tool *dotTrace*[2], which is also a .NET profiler. While dotTrace provides the necessary functionality, it stores its logs in a proprietary format, which made us unable to easily write signatures for the output. Therefore, we looked for an open-source alternative that provided similar functionality, which in turn led us to GroboTrace. The choice for a profiler was made as it can capture all calls made to the .NET CLR, which is precisely what is required to analyze the behaviour of an application. This includes calls from functions defined in the application and in the Framework Class Library, which means every managed call is captured. The .NET CLR is also the highest level layer shared by different .NET applications, making this an ideal place to capture the calls, as the output will be in the same format for every .NET application.

GroboTrace was initially used as-is to provide the necessary functionality. GroboTrace works by injecting callbacks into the functions as they are being compiled by the just-in-time compiler of the CLR. Additionally, it also injects the logger itself into the profiled application. While this is acceptable when profiling benign applications of which the behaviour can be controlled, it is not for malware. Malware might terminate unexpectedly or terminate non-graciously. GroboTrace would not be aware of this and be unable to save the logs before being terminated. Furthermore, GroboTrace also only injects its callbacks into functions outside of the Framework Class Library. This in turn means that calls to .NET Framework API functions will not be logged, making us unable to create signatures based on these functions. This is most likely done because the callbacks depend on functions in the Framework Class Library, which have to be JIT-compiled before they can be used. The JIT-compilation can however only finish after the callbacks have been injected. This would therefore result in a cyclic dependency when trying to inject callbacks into functions of the Framework Class Library. This again is not an issue when profiling the behaviour of a benign application, where the interest is generally in functions defined in the application itself. For the use case of this research this however does not work.

Instead of injecting callbacks like GroboTrace does, the *FunctionEnter, FunctionLeave*, and *FunctionTailcall* callbacks

[2]https://www.jetbrains.com/profiler/

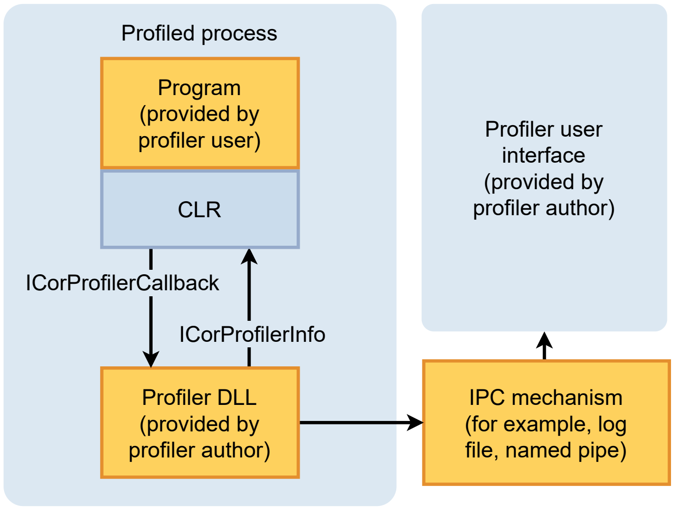

Profiled process
Program (provided by profiler user)
CLR
ICorProfilerCallback
ICorProfilerInfo
Profiler DLL (provided by profiler author)
Profiler user interface (provided by profiler author)
IPC mechanism (for example, log file, named pipe)

Fig. 6. Overview of how the application, profiler, and the interface (server) components interact [33]

of the profiling API were used. These callbacks allow us to monitor every function that is executed, and give us the required information to build the call tree. The downside of this is that these callbacks have to be implemented in C++, while GroboTrace was written in C#. This slightly increases the complexity of the tooling used as there are now two separate components implemented in different languages.

In order the tackle the issue of unexpected shutdowns, a separate server application to aggregate the calls has been developed as well. This application uses the output formatting of GroboTrace to produce the output logs. When the profiled application terminates, the server can still continue running and save the logs to disk. This allows the captured calls to be stored, no matter what happens to the profiled application.

As the functionality of the profiler has now been split into two separate instances, some form of communication between the applications is required. This is also what Microsoft recommends in their documentation, as can be seen in Figure 6. By separating the profiler into the profiler implementation and the log aggregator, the performance impact on the profiled application can be reduced. The calls are sent over a named pipe from the profiler to the server, which then adds the calls to the call tree.

The results of this process is a profiler which outputs every call made by a .NET application. This output can then be analyzed in order to extract the used techniques and to develop signatures for these techniques. A sample of the profiler output can be seen in Figure 7. In the sample it can be seen exactly which calls are involved in adding an entry to a hashtable in C#. Additionally, the sample also shows the structure of the call tree, where a call made by another function increases the depth. In the sample, the `Insert` function calls the `InitHash` function, which then calls three other functions itself.

It should be noted that calls in the profiler log are aggregated. This means that calling functions `A, A, B, A` in that order will show up in the logs as `3 calls to A, 1 call to B`. Some calls can appear in the logs hundreds of thousands of times, and storing these separately would increase the file-size from a few megabytes to potentially terabytes. This means that some detail is lost, but for the purposes of this research this is still sufficient as the function names themselves are what the signatures are based on.

Aside from the profiling implementation, we also briefly considered manually reverse engineering the malware samples. .NET Framework executables can be decompiled to C# code, which allows us to inspect their behaviour. The decompiler might however simplify or alter the output to make understanding easier, or be unable to decompile some functions at all. This could therefore influence our results, as we would like the behaviour to be the exact same as when the executable is executed on a machine. Furthermore, code obfuscation is very common in applications to hide what the executable does or to protect its code. In malware samples, code obfuscation is sometimes also used to avoid detection by anti-malware solutions. In case obfuscation is used, it would significantly increase the time it takes to analyze a file, depending on the level of obfuscation used. Additionally, we want to be able to analyze PowerShell scripts as well, as they are also part of the .NET Framework. These scripts are however written in a very different format from other .NET Framework executables, and we would therefore have to manually compare these formats. Manual comparison quickly becomes infeasible when there is a lot of malware to be compared, usually involving some kind of obfuscation. Additionally, one of the requirements of the tooling mentioned in Section III was that the output of the tooling should be the same regardless of which application it was executed on. Due to these major disadvantages, we decided to instead opt for a more general approach, which led us to the profiler.

In addition to the profiler, we also developed tooling that allows YARA rules to be applied to the profiler output. The choice for signatures in the form of YARA rules was made as YARA is a very popular pattern matching tool for malware. This allows other developers to easily write rules for our tooling, and allows existing signatures to be applied to our output with minimal changes. YARA rules, however, do not take into account the context in which the pattern occurs when searching for matches. This means that YARA can match function calls across different threads within the process, which could result in false positives. As the order and context in which the calls are used matter when attempting to write signatures for specific behaviour, we developed a Python wrapper script to make this possible. The Python wrapper script works by looking for a string called `$start` in the YARA rule, and searches the call tree starting from instances of that call. When the scope closes, the YARA rule is applied to all calls that occurred from the `$start` call to the end of the scope. This means that the YARA rule is now being applied to individual scopes, instead of the entire file. As a result, the Python script allows us to utilize the popular YARA format on our output logs, regardless of YARA not being context aware.

```
0.00% 0.061ms 1 calls System.Collections.Hashtable.Add
0.00% 0.061ms 1 calls System.Collections.Hashtable.Insert
    0.00% 0.061ms 1 calls System.Collections.Hashtable.InitHash
        0.00% 0.061ms 1 calls System.Collections.Hashtable.GetHash
        0.00% 0.061ms 1 calls System.Collections.CompatibleComparer.GetHashCode
        0.00% 0.061ms 1 calls System.String.GetHashCode
0.00% 0.061ms 1 calls Boo.Lang.Parser.BooLexer.Initialize
```

Fig. 7. Excerpt of the profiler log, showing the calls made when adding an entry to a C# Hashtable

B. *Profiler performance*

Initially the profiler performance was quite poor. An application which would take seconds to execute under normal circumstances could take over 2 hours to execute while being profiled. As this is not acceptable, the profiler had to be profiled in order to find out where potential performance improvements could be made.

1) Caching function names: Whenever a call was made in the application, the profiler would send a packet to the server. The packet would contain the ID and name of the function, the name of the module, the thread ID in which it was executed, and whether the function started or ended. Every time this occurred, the profiler would use the profiling API to request the function metadata to obtain all of this information. It turns out that this is a very time consuming action, and not necessary to be performed every time. Instead, a mapping from function ID to metadata was kept. The first time a function is called, its metadata is stored in a map. Every consecutive call would then look up the metadata in this map instead of using the API functions. While this increased the performance by over 75% (the application used for testing went from a runtime of 2 hours to 30 minutes), there was still room for more improvements.

2) Adding "map" packets: The metadata and IDs of functions in the .NET Framework API stay the same during the entire runtime of the application. Therefore, it is not necessary to send all of the metadata every time a function is called. Instead, only the first time a function is called its metadata is sent to the server using a `MAP` packet. The server will then keep track of the metadata for each function ID instead. Additionally, while implementing this performance improvement a better method of retrieving the function metadata was found. This method uses a different profiler API call, resulting in fewer calls made in total. These two additions brought down the runtime of the application by another 50% (the runtime went from 30 minutes to 15 minutes).

3) Adding a packet buffer: While looking for the most time consuming operations in the profiler, it was found out that almost half of the remaining time spent per call was in the communication between server and client. A new packet was created and sent for every call made, which results in many `WriteFile` calls being made to send this data over the named pipe. These calls were very time consuming compared to the rest of the calls in the profiler. Therefore, a buffer was made to allow multiple packets to be sent at once. Instead of sending every packet separately, the packets are appended to the buffer. Once the buffer exceeds a preconfigured size, the data of the buffer is sent to the server using a single `WriteFile` call, after which the buffer is emptied. This performance improvement increased the performance by another 50% (the runtime went from 15 minutes to 7 minutes).

All of these performance improvements brought the runtime of the profiled application down significantly. While the profiler still has an impact on the runtime of the application, it is fast enough to provide results for the analysis of this research.

The source code of the profiler has been made available on GitHub[3] under the MIT License.

VI. Evaluation

In this section, we present the results that we achieved by utilizing our profiler on fileless malware, using the steps described in Section IV. Following these steps, we created several signatures to identify the fileless malware techniques used in the analyzed samples. First, we describe the dataset we used and the setup in which we performed the experiments. Next, we evaluate our profiler and discuss the signatures that were created, after which we compare the signatures in more detail. During the comparison, we divide the signatures into distinct categories, based on the behaviour of the calls in the signatures. These categories in turn show which techniques for fileless malware for the .NET Framework we discovered during our analysis. After this distinction between techniques is made, we compare the behaviour of these techniques to uncover their similarities and differences. Special attention is paid to the DLR-based fileless malware technique in order to uncover exactly what separates it from existing techniques from an analysis standpoint.

A. Dataset & Experimental Setup

We profiled the samples on a virtual machine running Windows 10 in VirtualBox. Microsoft .NET Framework 4.8 was installed in order to run the profiler, server, and malware samples. As some of the frameworks used also have dependencies of their own, these were installed as well during the setup process of these frameworks.

Before going into the analysis process, we first describe our dataset and setup in more detail. We analyzed the following samples:

- Covenant v0.6
- SharpSploit v1.6
- Cobalt Strike v4.2
- SILENTTRINITY v0.4.6
- SILENTTRINITY v0.1.0

The samples listed are post-exploitation frameworks for the .NET Framework. These frameworks were selected as they are among the most well known frameworks and have been found used in malware. Additionally, these frameworks utilize fileless malware techniques themselves. By analyzing these frameworks and creating signatures for their techniques, we established a baseline of the current fileless malware techniques. The sample we analyzed that utilizes the DLR was SILENTTRINITY [12]. Two versions of SILENTTRINITY were analyzed as they both embed an interpreter in a different way. The older version, 0.1.0, utilizes the DLR to achieve this, while the most recent version, 0.4.6, uses a different technique instead. As SILENTTRINITY is a post-exploitation framework, adding other post-exploitation frameworks to the dataset allowed us to compare both DLR and non-DLR frameworks. This was done to create a better comparison between the different fileless malware techniques. These frameworks can be considered the state-of-the-art for .NET malware, and malware developers often use scripts from these frameworks [16], [18], [19], [46].

In order to analyze the techniques utilized by the frameworks, a stager for these frameworks had to be generated first. As the process of setting up the frameworks varies wildly, we describe this process individually for each framework. After this setup process, the dataset consisted entirely of files that could be executed and profiled directly.

Sample Configuration

1) Covenant: The first malware sample that we analyzed was the stager of the Covenant framework. As Covenant is an entire command and control framework, it requires the user to set up a server first. The author of Covenant has provided an in-depth installation and setup guide that can be followed to setup Covenant without any difficulty. The version that we selected for this research is the dotnet core version, which requires the user to simply clone the Covenant repository and execute the `dotnet run` command. After this, the user can navigate to the website shown in the dialog, and start using Covenant.

In order to create the Covenant stager (called launcher in Covenant), the user has to create a new listener first. This can be done by navigating to the listeners tab and creating a new listener with the default configuration. Next, the stager can be generated. For this research, we created a stager of type *binary*, as this generates a .NET Framework binary, which is the target of this research. When generating the stager it is important to set the *DotNetVersion* option to *Net40*, as this is what the profiler works with. When using the *Net35* version, the profiler will not be able to profile the binary and no logs will be captured. After this, the user will be given an executable file that can be used for analysis in the profiler.

2) SharpSploit: SharpSploit is a post-exploitation *library*, and technically not a framework like Covenant and Cobalt Strike. This means that another application can utilize the functionality provided by SharpSploit. One of these applications is in fact Covenant, which was developed by the same developer as SharpSploit. Many of the tasks that can be executed by the Covenant stager are provided by SharpSploit. Therefore, the configuration process of SharpSploit is the same as that of Covenant. The only difference is that in order to use SharpSploit functionality, a task utilizing SharpSploit has to be selected in Covenant. For this research, we selected the `Mimikatz`, `Rubeus`, `PowerShell` and `AssemblyReflect` tasks. As SharpSploit is an open-source project, the source code of the techniques it implements can

[3]https://github.com/oplosthee/dotnet-profiler

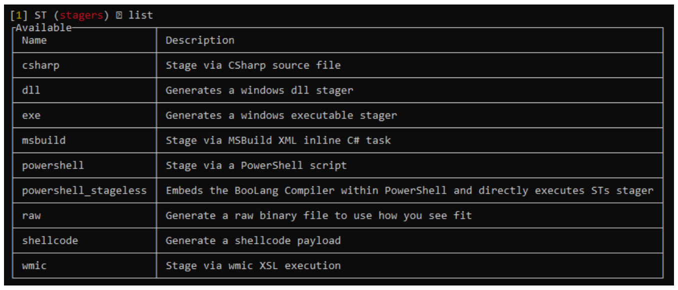
be found online.[4] In cases where a technique was not used by a Covenant task, we created its signature from the source code instead.

3) Cobalt Strike: The next malware sample that required configuration was Cobalt Strike. Cobalt Strike functions similarly to Covenant, but primarily focuses on unmanaged executables instead. It is however possible to generate a PowerShell and C# payload in Cobalt Strike, which is what we did for this research. We set up the Cobalt Strike teamserver on a Windows 10 virtual machine, using the Windows Subsystem for Linux. Next, we generated the stagers by selecting the Payload Generator under the Packages entry of the Attacks menu. The user is also prompted in the payload generator to create a listener if they have not done so already. For the listener we used the default settings, which results in an HTTP Beacon listener.

As the output formats, PowerShell, PowerShell command, and C# were selected. When inspecting the files however, it can be seen that the C# output merely results in a byte array containing shellcode. It is left to the user to actually load this shellcode into memory and execute it, which is what the primary interest for this research is. Therefore, no analysis could be performed on the shellcode. SharpSploit however does contain techniques to execute shellcode, so some methods to achieve this will still be considered for this research. The PowerShell command output behaves the same as the PowerShell output, but encoded into a single command instead of a script. For the analysis and signature creation phases we only considered the PowerShell output, as this contained the code for loading the malware into memory.

4) SILENTTRINITY: The last sample that required configuration was SILENTTRINITY. As was mentioned earlier, SILENTTRINITY is a post-exploitation framework, just like Cobalt Strike and Covenant. Therefore, it also requires the user to setup a server and generate a stager. To set up SILENTTRINITY v0.4.6, we followed the *"Installing from Source"* instructions that can be found on the wiki on GitHub. At the time of writing, there were some errors present in the code that made us unable to compile SILENTTRINITY from the master branch. A fork by a different user[5] that fixes these errors was used instead. These fixes do not change the behaviour of SILENTTRINITY in any way, and therefore also do not influence the results. After having compiled the fork, the instructions on setting up a server and client on the wiki can be followed. The configuration process for version 0.1.0 is the exact same, but the `legacy` branch was used instead of the `master` branch.

After starting the server and connecting a client to it, we could generate the stagers used for analysis. Similar to Covenant and Cobalt Strike, we had to create a listener first. In order to do this, we entered the `listeners` command, followed by the `use http` command to setup an HTTP listener. This listener was then started by using the `start`



```
[1] ST (stagers) ⊡ list
┌Available──────────────────────────────────────────────────────────────────────────────────────────────┐
│ Name                 │ Description                                                                    │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ csharp               │ Stage via CSharp source file                                                   │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ dll                  │ Generates a windows dll stager                                                 │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ exe                  │ Generates a windows executable stager                                          │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ msbuild              │ Stage via MSBuild XML inline C# task                                           │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ powershell           │ Stage via a PowerShell script                                                  │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ powershell_stageless │ Embeds the BooLang Compiler within PowerShell and directly executes STs stager │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ raw                  │ Generate a raw binary file to use how you see fit                              │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ shellcode            │ Generate a shellcode payload                                                   │
├──────────────────────┼────────────────────────────────────────────────────────────────────────────────┤
│ wmic                 │ Stage via wmic XSL execution                                                   │
└──────────────────────┴────────────────────────────────────────────────────────────────────────────────┘
```

Fig. 8. Overview of the stagers available in SILENTTRINITY

command. Depending on the system, it might be necessary to use a port higher than 1024 using the `set Port` command.

The next step was to generate the actual stager used for analysis. The stager configuration section can be accessed by entering the `stagers` command. SILENTTRINITY has multiple ways of staging the malware on a device. An overview of all of the stagers provided by SILENTTRINITY can be seen in Figure 8.

For this research, we used the `exe`, `powershell`, and `powershell_stageless` options. Unlike the other options, these stagers work out of the box and do not rely on another process to start them. We selected the stagers using the `use` command, and then generated the actual samples using the `generate http` command. The result of this is that the stagers will be saved to the directory of SILENTTRINITY, allowing us to analyze them using the profiler.

In order to execute an interpreted payload on the system and capture DLR related calls, we executed a built-in payload of SILENTTRINITY using the stager. We selected the `boo/msgbox` and `ipy/msgbox` modules as they are small modules, and create recognizable calls in the profiler output as the stager itself does not use message boxes. Therefore, the calls in the profiler log which open message boxes would allow us to pinpoint where the execution of the payload occurs in the log. The modules were selected by entering the `modules` command, followed by the `use boo/msgbox` or `use ipy/msgbox` command. The modules were then executed using the `run` command.

Profiler Configuration

Aside from the malware samples, the profiler and server have to be configured as well. Setting up the profiler is a fairly straightforward process. As the profiler is based on GroboTrace, the instructions for setting up GroboTrace apply. These instructions can be found in the readme file for GroboTrace.

The server requires no additional configuration and can be used right out of the box. Output logs generated by the profiler are stored in the same directory as the server executable itself. It should be noted that the logs are stored as `Log.txt`, and will overwrite previous logs if they are present in the directory.

As is mentioned in the readme for GroboTrace, the name of the process that should be profiled needs to be entered in the GroboTrace configuration file. In case the user wants to

[4]https://github.com/cobbr/SharpSploit/tree/master/SharpSploit/Execution
[5]https://github.com/d-sec-net/SILENTTRINITY

analyze a PowerShell script, `powershell.exe` should be added to this file. The profiler will then attach itself to any instance of PowerShell starting after this entry was added. Next, in order to profile a PowerShell script, the server should be started by simply running the executable. Then, when a PowerShell process starts, the profiler will automatically connect to the server and the logging process will start. After the PowerShell process terminates, the server will store the logs in the aforementioned `Log.txt` file.

The next step is to collect the logs, after which we can analyze them to find out which calls contribute to the fileless behaviour.

B. *Profiling*

One of the goals of this research was to develop a method that allows for different .NET applications to be compared, regardless of their programming language. As we showed in Section V, we achieved this goal with the development of our profiler. We successfully applied the profiler to the samples in our dataset, which include both executables written in C# and PowerShell scripts. For all of these instances, the profiler correctly produced an output log containing the call tree of the sample under analysis. In order to verify that our method is indeed applicable to any .NET Framework application, we applied it to the languages officially supported by Microsoft. This includes the C#, F#, and Visual Basic programming languages [3]. We wrote a simple program that writes a list of numbers to the console in all three languages and profiled the executables. The resulting logs were all very similar, with some minor changes between the three languages. The startup procedure was the same for all three samples, and can be seen in Figure 9.

```
0.00% 0.000ms 1 calls System.AppDomain.SetupDomain
0.00% 0.000ms 1 calls System.AppDomain.InitializeCompatibilityFlags
0.00% 0.000ms 1 calls System.CompatibilitySwitches.InitializeSwitches
0.00% 0.000ms 1 calls System.AppDomain.InitializeDomainSecurity
0.00% 0.000ms 1 calls System.AppDomain.TurnOnBindingRedirects
0.00% 0.000ms 1 calls Microsoft.Win32.SafeHandles.SafePEFileHandle..ctor
0.00% 0.000ms 1 calls System.Security.Policy.PEFileEvidenceFactory.CreateSecurityIdentity
0.00% 0.000ms 1 calls System.Security.Policy.AssemblyEvidenceFactory.UpgradeSecurityIdentity
0.00% 0.000ms 1 calls System.AppDomain.SetTargetFrameworkName
```

Fig. 9. The startup procedure shared across samples in different languages

The calls that follow the startup procedure involve functions to enumerate the list in both Visual Basic and F#, but have most likely been optimized out in C#, as only a `WriteLine` call is present. The profiler logs for the execution of our program in C#, F#, and Visual Basic can be found in Figure 10, Figure 11, and Figure 12 respectively.

```
0.00% 0.000ms 1 calls CSharpSample.Program.Main
    0.00% 0.001ms 5 calls System.Console.WriteLine
```

Fig. 10. The execution of the test sample in C#

As can be seen in the output logs, our method was successfully able to profile samples in multiple languages for the .NET Framework. This makes sense, as all .NET languages compile to the Common Intermediate Language, which is then executed by the Common Language Runtime. As our profiler uses the profiling API of the Common Language Runtime, we can therefore profile any application that compiles to the Common Intermediate Language. The output logs are also all in the same call tree format, making it possible to compare the different logs to each other. These experiments in turn show that our profiler was indeed an answer to our first research question: how can different .NET Framework applications be compared, regardless of their programming language?

```
0.00% 0.000ms 1 calls Program.main
    0.00% 0.000ms 1 calls Microsoft.FSharp.Collections.FSharpList`1..cctor
    0.00% 0.000ms 1 calls Microsoft.FSharp.Collections.FSharpList`1.get_Empty
    0.00% 0.001ms 5 calls Microsoft.FSharp.Collections.FSharpList`1.Cons
    0.00% 0.001ms 6 calls Microsoft.FSharp.Collections.FSharpList`1.get_TailOrNull
    0.00% 0.001ms 5 calls Microsoft.FSharp.Collections.FSharpList`1.get_HeadOrDefault
    0.00% 0.001ms 5 calls System.Console.WriteLine
```

Fig. 11. The execution of the test sample in F#

```
0.00% 0.000ms 1 calls VBSample.Module1.Main
    0.00% 0.000ms 1 calls System.Collections.Generic.List`1..ctor
    0.00% 0.000ms 1 calls System.Collections.Generic.List`1.GetEnumerator
    0.00% 0.001ms 6 calls Enumerator.MoveNext
    0.00% 0.001ms 5 calls Enumerator.get_Current
    0.00% 0.001ms 5 calls System.Console.WriteLine
    0.00% 0.000ms 1 calls Enumerator.MoveNextRare
    0.00% 0.000ms 1 calls Enumerator.Dispose
```

Fig. 12. The execution of the test sample in Visual Basic

We additionally specified two requirements that the method had to adhere to, namely being deterministic and producing a programmatically parsable output. Both of these requirements have been fulfilled and tested during the signature creation phase of our approach. In order to verify our signatures, we ran the malware samples under the profiler an additional time, producing a new output log. In all tested cases, the signature matched on both the output log it was created from, and the new output log used for verification. This shows us that the profiler did indeed produce a deterministic log, making it usable for this research. Furthermore, the fact that we were able to write signatures for the output and apply these using YARA show that the output is also indeed programmatically parsable.

C. *Analysis & Signatures*

The second research question this research aimed to answer was: what are the current fileless malware techniques for the .NET Framework? In order to answer this research question, we applied our profiler to the dataset, following the steps described in Section IV. After having used the profiler to collect call logs for the samples in the dataset and having analyzed the relevant calls belonging to each technique, we created signatures in the form of YARA rules for the techniques. The signatures each belong to a specific technique, therefore also showing us what the current techniques are, and thus answering our second research question. Additionally, as these signatures capture the characteristics of each technique, they allow for an excellent way to compare the different techniques. This in turn also allows us to answer our third and final research question, which is to analyze the differences and similarities between fileless malware techniques.

With the exception of SharpSploit, every sample in the dataset has one unique signature. SharpSploit, however, has multiple signatures, as SharpSploit is a post-exploitation library. SharpSploit implements multiple fileless malware techniques, and allows developers to use these techniques in their own malware. As was mentioned earlier in this section, we used several Covenant tasks that implement these techniques to create multiple call logs for SharpSploit. As a result of this process, SharpSploit has multiple signatures belonging to the different techniques. All of the signatures have been made available on GitHub[6].

In order to answer our second research question, we examined the signatures in more detail to understand the .NET API calls used to implement the techniques. This was done to group techniques using similar calls together, allowing us to create an overview of distinct techniques. This overview will in turn show us the current fileless malware techniques for the .NET Framework. When comparing the signatures, a clear distinction can be seen between techniques that use unmanaged code and those that do not. Unmanaged code is code that is not executed by the .NET runtime, and is therefore also not profiled by the profiler. While this does mean we do not have full insight into the behaviour of the technique, we can still see how the unmanaged code is called in the call logs. As even these techniques still use some managed code, signatures could still be written for them.

The techniques we found that did not use unmanaged code were all based around reflection, with the exception of the sample utilizing the DLR. Reflection in programming refers to the ability of a program to inspect and modify itself, which in the context of fileless malware allows the program to add new, malicious code to the application [34]. In the .NET Framework, this is mostly achieved through the `System.Reflection` namespace. The basis of this technique is the `Assembly.Load` function, which loads an application into the memory of the application that calls it, given the contents of the new application in bytes. The bytes can be downloaded from a remote source and the bytes can exist entirely in memory, making this function applicable for fileless malware. This technique can be found used in both Covenant and SharpSploit. The signature that we made for this technique for the two samples can be found in Figure 13.

While SharpSploit has several implementations of this technique, they all involve the same two core function calls: `Assembly.Load` to load the payload, and `RuntimeMethodInfo.Invoke` to execute it. These calls can be seen in Figure 13 as well, as the strings `$start` and `$x4`. An alternative implementation of SharpSploit of this technique switches out the `GetMethod` calls for a `get_EntryPoint` call. The `get_EntryPoint` call simply returns the entry point of the application as a method, similar to what `GetMethod` would do. The difference is that `GetMethod` allows the user to specify the method by name. Another alternative implementation that is present both in SharpSploit and Covenant is prefixed by a call to the `System.Convert.FromBase64String` function.

[6]https://github.com/oplosthee/profiler-rules

```
rule SharpSploit_AssemblyExecute_Parameters
{
  strings:
    $start = "System.Reflection.Assembly.Load"

    // Loaded assembly is executed on the same thread (
        singular and plural):
    $xa2 = "System.Reflection.Assembly.GetType"
    $xb2 = "System.Reflection.Assembly.GetTypes"
    $xa3 = "System.Type.GetMethod"
    $xb3 = "System.Type.GetMethods"
    $x4 = "System.Reflection.RuntimeMethodInfo.Invoke"

    // Loaded assembly is executed on a new thread:
    // This does not occur in SharpSploit, Covenant
        however can do this.
    $y2 = "System.Threading.Thread..ctor"

  condition:
    all of ($s*) // Assembly.Load
    and
    (
      (
        (
          all of ($xa*) // Singular methods
          or
          all of ($xb*) // Plural methods
        )
        and
        $x4 // The invoke in combination with singular
            OR plural methods
      )
      or
      all of ($y*) // New thread creation after Load
    )
}
```

Fig. 13. YARA rule created for the detection of reflection-based techniques

As the name implies, this decodes a base64 encoded string, which would contain the malicious payload in the case of fileless malware. The payload is then loaded in the same way as shown in the signature. The base64 encoding is done to obfuscate the payload of malware, making it slightly harder to detect when static analysis is used to detect specific structures in the payload. Another advantage of using base64 encoding is that it can make it easier to transfer the payload, as the payload can be represented as text instead of a binary file after encoding. While this fileless malware technique can work when the payloads are undetected by anti-malware solutions, it is not perfect. As was mentioned in Section II-F, AMSI scans all loaded assembles since .NET 4.8. This includes assemblies loaded through the `Assembly.Load` call we saw being used in these techniques. This technique for fileless malware is therefore theoretically as easily detectable as it would be without using fileless malware techniques, as the payload is scanned for malware either way.

Surprisingly, we also discovered usages of reflection in the SILENTTRINITY sample. Unlike the instructions included with SILENTTRINITY indicate, the most recent version does in fact not use the .NET DLR. This version of SILENTTRINITY does still embed an interpreter, but it embeds the Boo programming language instead of IronPython [45]. Boo is a language for the .NET Framework, using a Python-like syntax instead of C# [9]. The Boo interpreter was released before the DLR was available, and therefore does not use this component, unlike IronPython. Older versions of SILENTTRINITY do include IronPython, which meant that we also had to test an older version of the malware. This older version did indeed use the .NET DLR, which we determined from the presence of calls in the `Microsoft.Scripting` namespace. The newer sample we analyzed instead uses reflection in the form of dynamic methods to load and execute the payloads. Dynamic methods allow developers to compile and execute methods at runtime [29], which is done for the Boo payload in SILENTTRINITY. This behaviour can be seen in Figure 14. The embedded interpreter first processes the payload, similarly to a compiler, resulting in a dynamic assembly. The processing of the payload was omitted from the signature, as it includes thousands of calls for lexing, parsing and code generation functionality, while we are only interested in the resulting assembly. This assembly is then invoked and the dynamic method is called, resulting in the payload being executed. In the next section, where we compare the techniques in more detail, the differences between the DLR-interpreter and the non-DLR-interpreter are covered in more detail.

```
rule SilentTrinity_BooRuntime
{
    strings:
        $start = "Boo.Lang.Compiler.CompilerContext.
            get_GeneratedAssembly"
        $s1 = "Boo.Lang.Runtime.RuntimeServices.Invoke"
        $s2 = "System.Reflection.Emit.DynamicResolver"

    condition:
        all of them
}
```

Fig. 14. YARA rule created for the detection of the Boo runtime in SILENTTRINITY

On the other hand we found fileless malware techniques utilizing unmanaged code as opposed to the reflection calls seen in the managed techniques. While our profiler was not able to profile the unmanaged functions that were executed, we still had some insight into the managed part of these techniques. These techniques often involve marshaling, which is the process of converting types between their managed and unmanaged variants. An example of this would be an `int64_t` in native code being converted to a `long` in a managed language like C#. This process can also occur when unmanaged functions are called from managed code. In this case, the `System.Runtime.InteropServices.Marshal.GetDelegateForFunctionPointer` function is called by the managed code. In other cases, where the unmanaged code is called through unmanaged functions, we unfortunately cannot see this process in the call logs. The inability to track unmanaged functions is a shortcoming of the profiler, and is addressed in Section VIII. Nevertheless, in the unmanaged fileless malware techniques where the marshaling calls are present, signatures can still be made using these calls. This was the case for a technique used in SharpSploit, for which we created a signature. This signature can be found in Figure 15.

```
rule SharpSploit_ShellCode
{
    strings:
        $start = "System.Runtime.InteropServices.
            GCHandle.Alloc"
        $s1 = "System.Runtime.InteropServices.GCHandle.
            AddrOfPinnedObject"
        $s2 = "System.Runtime.InteropServices.Marshal.
            Copy"
        $s3 = "System.Runtime.InteropServices.Marshal.
            GetDelegateForFunctionPointer"

    condition:
        all of ($s*)
}
```

Fig. 15. YARA rule created for the detection of P/Invoke-based techniques

This signature detects shellcode being copied to memory, which is then executed. When comparing the signature to the code of SharpSploit on GitHub[7], it can be seen that the `Kernel32.VirtualProtect` call is not present in the signature. This is correct, as the `VirtualProtect` function is an unmanaged function, and therefore not present in the profiler output.

The .NET Framework feature that makes this possible is called Platform Invoke (P/Invoke) [32]. P/Invoke allows managed applications to call into unmanaged libraries, such as `kernel32.dll`, where the `VirtualProtect` function is defined. By adding the `DllImport("kernel32.dll")` attribute to a managed method with the same signature, the runtime knows it has to load the specified DLL and that the function is defined there. Calling this method will then in turn call the unmanaged function. We also found another technique capable of achieving this behaviour implemented in SharpSploit. This technique is called *Dynamic Invocation* by the SharpSploit developers, or D/Invoke for short. D/Invoke allows for the same fileless malware possibilities as P/Invoke, but does not require the `[DllImport]` attribute like P/Invoke does. Therefore, the DLLs and functions are also not present in the metadata of the .NET assembly, avoiding suspicious function calls when statically analyzing the application [49]. We thus consider these two techniques as separate techniques, even though they appear similar from their behaviour.

Now that we have compared the signatures in more detail, we can make an overview of the current fileless malware techniques for the .NET Framework and answer our second research question. We divide the behaviour seen in the signatures into distinct categories, each representing a fileless malware technique. The result of this will be a list of the current fileless malware techniques for the .NET Framework, after which we discuss their similarities and differences in more detail.

TABLE I
TYPES OF TECHNIQUES IMPLEMENTED BY EACH ANALYZED SAMPLE

Sample	**Technique**
Covenant (stager)	Reflection-based
SharpSploit (PE loading)	P/Invoke-based
SharpSploit (ShellCode)	P/Invoke-based
SharpSploit (.NET Assembly)	Reflection-based
SharpSploit (Generic)	D/Invoke-based
SharpSploit (Injection)	D/Invoke-based
Cobalt Strike	D/Invoke-based[8]
SILENTTRINITY v0.1.0 (PowerShell)	Embedded interpreter
SILENTTRINITY v0.4.6 (Executable)	Embedded interpreter
SILENTTRINITY v0.4.6 (PowerShell)	Embedded interpreter
SILENTTRINITY v0.4.6 (PowerShell (Stageless))	Embedded interpreter

D. *Fileless Malware Techniques*

Now that we have analyzed the samples and identified their fileless techniques, we can answer our second research question: what are the current fileless malware techniques for the .NET Framework? In the overview in Table I, four distinct types of techniques can be seen: reflection-based, P/Invoke-based, D/Invoke-based, and a technique using an embedded interpreter. These types of techniques are what we discovered in our dataset, and therefore the kinds of fileless malware techniques for the .NET Framework we discovered, answering the second research question. This distinction between techniques was based on the functions that were used to implement the functionality, which we discussed in the previous subsection. The reflection-based techniques, for example, utilize functions from the `System.Reflection` namespace to load assemblies into memory and execute them. However, the signatures for these techniques do not necessarily have to be the exact same. An example of this would be the signatures for Covenant and the .NET assembly loading function of SharpSploit. Covenant has the ability to start the loaded assembly in another thread, while SharpSploit starts it on the same thread as the one that loaded the assembly into memory. This in turn results in slightly different profiler outputs between the two samples. In this section, we discuss the different techniques we encountered in more detail, including their differences and similarities. This in turn allows us to answer our third and final research question, which is to identify the differences and similarities in characteristics between the technique based on the DLR, and the other types of fileless malware techniques.

1) Reflection-based Techniques: The group of reflection-based techniques is the first group of techniques we encountered during analysis. The techniques in this group primarily utilize reflection, like was seen in SharpSploit and Covenant in the previous section. More specific, they all use the `Assembly.Load` function call, which loads an assembly into the same environment as the assembly that called the function. As a result of this, we can also see the calls made by the loaded assembly in our profiler logs.

Most commonly, the `Assembly.Load(Byte[])` variant of the function call is used. This function call lets the user load an application in the form of a byte array. This in turn means that the assembly does not have to be present on the system of the user, as the entire assembly is given as the argument. This is commonly used in malware, such as Covenant and SharpSploit, by making the malware download an assembly as bytes, which can then be loaded through the `Assembly.Load` function. After having loaded the assembly, it is possible to execute the functions defined in the assembly. The function which is to be executed has to be retrieved first, after which it can be invoked using the `MethodInfo.Invoke` function. It is possible to select any function defined in the assembly by manually specifying the name of the function in the `GetMethod` function. In case no function is specified, the entrypoint of the assembly is invoked.

The advantage of using this technique is that it is extremely simple for a developer to set up, as it simply involves two function calls to filelessly load and start the malware. Furthermore, these reflective calls are also commonly used in benign .NET Framework applications. Therefore, the presence of these calls does not directly indicate malicious behaviour. The biggest downside of this technique is that AMSI scanning has been added to all assemblies loaded through `Assembly.Load` in .NET Framework 4.8 [38]. This means that in case the malicious payload is detected by anti-malware solutions, the runtime will detect and stop the malware from being loaded. Depending on the anti-malware solution used, this also means that the execution of the malicious payload is now logged. As fileless malware techniques are commonly used to stay undetected, the addition of AMSI scanning is a significant downgrade to the effectiveness of this technique.

2) P/Invoke-based Techniques: The next group of techniques we encountered were techniques using Platform Invoke, or P/Invoke for short. P/Invoke allows users to access functions defined in unmanaged libraries [32]. This technique is present in the .NET Framework by default, similar to the reflection-based techniques. By itself, P/Invoke does not allow developers to load malicious payloads into memory, but the

[7]https://github.com/cobbr/SharpSploit/blob/eb58caaba734de9b18450dab493b41d5a9b5464e/SharpSploit/Execution/ShellCode.cs

[8]The approach is similar to the SharpSploit technique, however different methods are used to achieve it.

unmanaged functions that can be accessed through it can allow this.

As we showed during the analysis, this technique requires developers to add a `[DllImport]` attribute to a function in their managed code. The example from the SharpSploit sample we showed earlier calls the `VirtualProtect` function by adding the attribute to a managed function with the same signature as the `VirtualProtect` function. The techniques that utilize this functionality therefore combine managed code with unmanaged code in order to stay undetected. As our profiler only has insight into the managed calls, it means that we cannot see these unmanaged calls in the profiler logs.

An example of how this technique can be utilized to load a malicious payload into memory can be seen in SharpSploit. SharpSploit has the option to execute unmanaged shellcode in memory, using the P/Invoke technique. We have created a signature for this technique from the SharpSploit output logs, which can be seen in Figure 15. First, SharpSploit allocates memory for the shellcode using the `GCHandle.Alloc` function. Then, using the `Marshal.Copy` function, it copies the shellcode to the location of the memory that was allocated earlier. Until this point, only managed functions have been used, which show up in the logs of our profiler. The allocated memory is however not executable by default, but the `VirtualProtect` function makes it possible to change the permissions of the allocated memory. The `VirtualProtect` function is then called using P/Invoke to mark the allocated memory as executable. The only step left is to execute the shellcode in memory using the `Marshal.GetDelegateForFunctionPointer` function. This function is commonly seen in fileless malware techniques utilizing unmanaged code, such as P/Invoke. The function converts the pointer for the memory address that was allocated earlier to a delegate, allowing it to be executed from managed code. As we can see, putting this all together makes it possible to both load and execute a malicious payload in memory.

By combining both managed and unmanaged code, the P/Invoke-based techniques are able to avoid analysis with our tooling to a certain level. The unmanaged calls do not show up in the logs of our profiler, meaning we do not know which unmanaged functions were called. The marshaling functions do however show up, as they are managed functions themselves. The signatures we made to detect these techniques were therefore based on these marshaling calls, and still allowed us to detect these techniques regardless. Combining both managed and unmanaged code is what makes these techniques powerful, as targeted detection no longer works. Detection methods that specifically look at unmanaged code would be unable to detect malware in the managed code, and vice versa. This is also what we encountered during the analysis with our tooling.

The downside of this technique is however the requirement of the `[DllImport]` attributes, which give away which functions are being called when applying static analysis. The binaries that are imported by the attribute are present in the metadata of the application, specifically the `ModuleRef` table. When the application is loaded, the required DLLs are loaded into memory as well. This includes the DLLs specified with the `[DllImport]` attribute. This does however open up the opportunity for users to hook these functions. Hooking is the process of intercepting a function call by replacing it with a different function call. For SharpSploit, this means that the `VirtualProtect` function can be hooked by a detection platform, replacing it with a different function. This function could then be used to detect whether what is passed into `VirtualProtect` is malicious or not. This shows that while the P/Invoke-based techniques have the strong advantage of utilizing unmanaged code, this advantage comes with its own drawbacks.

3) D/Invoke-based Techniques: The third group of techniques we encountered were based on dynamically invoking unmanaged code. As the name of this category implies, this group of techniques behaves similar to the P/Invoke-based techniques in the sense that it allows developers to call unmanaged code. However, the D/Invoke-based techniques do not utilize the `[DllImport]` attribute that P/Invoke utilizes. Instead, the DLLs and the functions are resolved dynamically during runtime. This way there is no indication of the unmanaged functions that are called when applying static analysis to samples using D/Invoke.

In order to dynamically resolve a function, the address of the function is obtained from the export address table of the library. This table contains all the functions that the library exports, together with their addresses [31]. It is possible to retrieve this information given the base address of where the library was loaded, which is what D/Invoke does. After having retrieved the function pointer, it is possible to execute the function by executing the `Marshal.GetDelegateForFunctionPointer` function, followed by the invocation of the resulting delegate. This function is also used in SharpSploit and Cobalt Strike to execute shellcode. It should be noted that retrieving the functions from the export address table only works when the library has already been loaded by the application. In case the developer wants to call a function defined in `kernel32.dll`, such as the `VirtualProtect` function we saw earlier, it has to be loaded first. This would usually be done through the `LoadLibrary` function, but as this is an unmanaged function from an unloaded library, P/Invoke would be required to call it. Instead, the authors of D/Invoke cleverly utilize the `LdrLoadDll` function from the `ntdll.dll` library [49]. This library is loaded into every process on Windows, and therefore will also be loaded in our .NET Framework application. From this library the `LdrLoadDll` function is then called using the address in the export table, similar to how other unmanaged functions would be executed. Using this process, it is possible to call unmanaged code from managed code, without using the `[DllImport]` attribute like P/Invoke.

When analyzing a sample utilizing a D/Invoke-based technique with our profiler, we are only able to see the marshalling calls, similar to P/Invoke-based techniques. Therefore, our signatures are also based on these calls. In order to find the export address of a library function, many marshalling calls have to be made as unmanaged code cannot be used. This process can be seen in the source code of the D/Invoke implementation of SharpSploit [50]. This information would make it possible to distinguish between P/Invoke and D/Invoke-based techniques when examining our profiler output.

In a blog post by the authors of D/Invoke, they mention that there are other methods to detect the usage of the technique. For example, ETW logs will be generated when loading DLLs with D/Invoke using the technique mentioned earlier. This makes it possible for anti-malware vendors to monitor these logs and detect the usage of D/Invoke-based techniques. The authors came up with methods to avoid this to a certain extent, one of which being manually mapping the libraries into memory. When manually mapping the library into memory, the `LdrLoadDll` call is not used. Instead, the library is read manually from the disk or from a byte array into the memory of the process. This makes the technique even more fileless than it already is, as the byte array can exist purely in memory. Additionally, this also makes anti-malware vendors unable to hook functions in the library. The manual mapping of libraries does however result in more possibilities of detection for our profiler as there are now more managed calls used by the D/Invoke technique.

4) Techniques using an Embedded Interpreter: The fourth and final technique we encountered were techniques utilizing an embedded interpreter. These techniques are also the techniques utilizing the Dynamic Language Runtime, which is what this research is specifically targeted at. The analysis of these techniques will therefore also allow us to answer the third research question: what are the characteristics of fileless malware techniques using the Dynamic Language Runtime? As was mentioned in the description of the dataset, two versions of SILENTTRINITY were analyzed, both including an embedded interpreter. The most recent sample, version 0.4.6, does not utilize the DLR while the older sample, version 0.1.0, does. Other than giving us insight into fileless malware utilizing the DLR, these two samples also make for a much better comparison of DLR-based techniques and non-DLR-based techniques. As the samples behave in the same way to the user, most of the differences in the profiler output will be due to the usage of the DLR in the older version.

On a high level, these techniques work by embedding an interpreter made for the .NET Framework into the application. By doing this, it is possible to write a malicious payload as a script for the language of the interpreter and have the interpreter execute the payload. This can be very convenient for malware developers, as they can write a script and send it to the infected machine for execution, without having to compile anything. This is comparable to a PowerShell script, where you can simply run the script as-is from the PowerShell command line. Furthermore, it is possible for the interpreted script to only exist in memory, hence it is part of this research.

While these techniques are able to bypass detection methods such as AMSI, as we mentioned in Section II, the actual implementation is fairly straightforward. The Boo compiler is available as a library that any developer can simply add to their .NET Framework project, which is also done in SILENTTRINITY. Then, the API functions of the compiler can be used to compile the Boo source code in memory, and execute it. This process is similar for IronPython, which is used in the older version of SILENTTRINITY. It should be noted that a Boo compiler is used, as opposed to an interpreter. This is due to the differences between DLR and non-DLR fileless malware, which we explain in more detail later in this section. For the developer this is transparent, as both approaches take a script as input and execute it. Until this point, both of the approaches to embedded interpreters appear to function in the exact same way. The implementation of the interpreters is however where the differences between the DLR and non-DLR variants are.

When profiling a sample of the newer version of SILENTTRINITY, which does not utilize the DLR, we can clearly see the steps taken by the Boo compiler. First, the input source is parsed. This is visible in the profiler output in the form of calls from the `Boo.Lang.Parser`, `Boo.Lang.Compiler.Ast` and `antlr` namespaces. ANTLR is a popular parser generator [40], which can create a parser for a language given the grammar for that language. This confirms that the Boo payload is indeed being compiled on the spot by SILENTTRINITY. A part of this process can be seen in the profiler output in Figure 16.

```
0.061ms 1 calls Boo.Lang.Parser.BooParsingStep.Run
    0.061ms 1 calls Boo.Lang.Parser.BooParsingStep.ParseModule
        0.061ms 1 calls Boo.Lang.Parser.BooParser.CreateParser
        0.061ms 1 calls Boo.Lang.Parser.BooParserBase.start
            0.061ms 1 calls Boo.Lang.Parser.BooParserBase.parse_module
                1.029ms 17 calls antlr.LLkParser.LA
```

Fig. 16. Excerpt of the profiler output showing the Boo compilation process

After the compilation has finished, we are presented with the logs on which the signature in Figure 14 is based. The presence of a compiler however indicates that the payload is actually not interpreted, unlike the author claims. As we can see in the profiler output in Figure 17 and from the compiler calls mentioned earlier, the payload is compiled and then invoked in its entirety. This profiler output belongs to the `boo/msgbox` payload, of which the source code can be seen in Figure 18.

```
0.061ms 1 calls Boo.Lang.Runtime.RuntimeServices.Invoke
    0.061ms 1 calls FlyZGLmT7OModule.Main
        0.061ms 1 calls System.Windows.Forms.MessageBox.Show
        0.091ms 1 calls System.Console.WriteLine
```

Fig. 17. Profiler output showing the `boo/msgbox` task being executed by SILENTTRINITY (runtime percentages omitted for brevity)

The non-DLR version of SILENTTRINITY, and therefore Boo, instead uses dynamic methods to achieve its fileless behaviour. As was mentioned earlier, dynamic methods allow developers to compile and execute methods at runtime [29].

```
import System.Windows.Forms as WinForms

public static def Main():
  WinForms.MessageBox.Show("WINDOW\_TEXT", "WINDOW\
      _TITLE")

  print 'Popped'
```

Fig. 18. The source code of the `boo/msgbox` of SILENTTRINITY

This is exactly what the Boo compiler does, as opposed to utilizing the DLR functionality.

This however brings us to the main difference between the two versions of SILENTTRINITY, and consequently the usage of the DLR for fileless malware. The first and most recognizable characteristic of the sample utilizing the DLR is the presence of the `Microsoft.Scripting` namespace. This namespace is a part of the DLR, and will therefore only be present in executables utilizing the DLR. Other than the namespaces used, we can also see the behaviour of the IronPython interpreter in our profiler logs. This is another characteristic of the DLR, as interpreting a dynamic language would not be possible without it, which was the case with the Boo compiler. Instead of compiling the payload first and then executing it, the payload is interpreted line by line. This means that unlike the newer SILENTTRINITY sample, the log output is completely different from what can be seen in Figure 17. When executing the `ipy/msgbox` payload, which can be seen in Figure 19, we can clearly see the differences between the two SILENTTRINITY versions. Instead of the function calls being in the same scope, they are completely separated from each other. The call to the `MessageBox.Show` function in the profiler logs can be seen in Figure 20. As can be seen, the function call is surrounded by DLR-specific calls made by the IronPython interpreter. In a different scope, hundreds of lines removed in the output from the first function call, we can see the `print` call, as shown in Figure 21.

```
import clr
clr.AddReference("System.Windows.Forms")
import System.Windows.Forms as WinForms

WinForms.MessageBox.Show(str("WINDOW_TEXT"), str("
    WINDOW_TITLE"))

print 'Popped'
```

Fig. 19. The source code of the `ipy/msgbox` of SILENTTRINITY

```
0.00% 0.107ms 1 calls Microsoft.Scripting.Interpreter.Interpreter.Run
    0.314ms 3 calls Microsoft.Scripting.Interpreter.BranchInstruction.Run
    0.107ms 1 calls Microsoft.Scripting.Interpreter.FuncCallInstruction`3.Run
        0.107ms 1 calls System.Windows.Forms.MessageBox.Show
```

Fig. 20. Profiler output showing the `WinForms.MessageBox.Show` part of the `ipy/msgbox` task being executed by SILENTTRINITY (runtime percentages omitted for brevity)

It can be seen that both of the function calls are surrounded by DLR-specific calls made by the interpreter. This is the most significant difference in characteristics between the DLR variant and the non-DLR variant of SILENTTRINITY. Additionally, this also makes it much harder to reverse engineer the functionality of the payload, as the contents of the payload are not grouped together. While it would be possible to apply differential analysis on the profiler logs to filter out the interpreter related calls, obfuscation techniques could be used to counter this. These differences also show us what the characteristics are of fileless malware techniques utilizing the Dynamic Language Runtime, answering our third research question. Additionally, the comparison with the non-DLR variant allowed us to answer the sub-question of this research question as well, which was to determine the differences and similarities between DLR and non-DLR based fileless malware techniques.

```
0.107ms 1 calls Microsoft.Scripting.Interpreter.ActionCallInstruction`2.Run
    0.107ms 1 calls IronPython.Runtime.Operations.PythonOps.Print
```

Fig. 21. Profiler output showing the `print` part of the `ipy/msgbox` task being executed by SILENTTRINITY (runtime percentages omitted for brevity)

As we have shown, of the two samples using an embedded interpreter, only the sample utilizing the DLR truly has an embedded interpreter. When compared to the other techniques, this technique is most similar to the reflection-based techniques, as both only involve managed code. This is especially true for the Boo variant, as the calls to generate the dynamic methods are also included in the `System.Reflection` namespace. Other than using purely managed code, the actual implementation of the IronPython variant is completely different from the Boo variant or any of the other techniques we have seen.

In conclusion, we encountered four types of fileless malware techniques for the .NET Framework. With this list of techniques we were able to answer our second research question, which was targeted at finding out the current techniques for fileless malware for the .NET Framework. Each of the techniques we encountered produced a distinct output when analyzed using the profiler we developed for this research. The types of fileless malware can be split into roughly two groups: the techniques that utilize unmanaged code, and those that do not.

The two types of techniques that utilized unmanaged code were the P/Invoke and D/Invoke-based fileless malware techniques. These techniques both called unmanaged code, making our managed profiler unable to get insight into the functions that were called. The methods they used to achieve this were however different for both techniques. As we showed, the samples utilizing the P/Invoke-based technique left a clear indication of the unmanaged functions that were used in the assembly metadata. On the other hand, static analysis on samples utilizing the D/Invoke-based technique did not show any indication of which unmanaged functions were called. On the managed side, which our profiler was able to analyze, the techniques however appeared to behave the same. Both techniques utilized managed marshaling functions to convert types between managed and unmanaged variants, allowing us to build signatures for the techniques. Additionally, the P/Invoke-based technique also opened up the possibility of function hooking, which the D/Invoke-based technique prevents by dynamically loading libraries.

The other two types of techniques, those that utilize managed code, were the techniques utilizing reflection and embedded interpreters. These techniques involved no unmanaged code at all, allowing us to get insight into the full functionality of the technique using our profiler logs. The reflection-based fileless malware technique utilized built-in functions to the .NET Framework, similar to the P/Invoke-based technique. This in turn also makes it susceptible to detection, as AMSI monitors the built-in functions used for reflection. This is not the case with the other techniques, which are much harder to detect. As we showed, depending on the implementation of the embedded interpreter, this technique also utilizes reflection. Therefore, these two techniques can be considered very similar from an analysis standpoint. This however changes when the Dynamic Language Runtime is utilized for embedding the interpreter. When using the DLR, the entire output of our profiler looks unlike that of any of the other samples. Instead, an actual interpreter built on top of the DLR is used to interpret the payload. This is also visible in our logs, as calls from the payload are surrounded in multiple lines of interpreter related calls. This in turn makes it much harder to analyze using our profiler, as it becomes harder to find the calls belonging to the payload. Additionally, the DLR-based fileless malware technique utilises calls from the `Microsoft.Scripting` namespace. This namespace is not present in the other profiler logs, as it is specific to the DLR. Other than the behaviour visible in the profiler logs, this namespace could be used to detect usage of the DLR. This information also answers our third and final research question, which was to find the characteristics of fileless malware techniques using the DLR, and how it differs from other kinds of fileless malware techniques for the .NET Framework.

VII. Limitations

While performing our analysis, we noticed several limitations of our approach to analyzing .NET executables. First and foremost, the most significant limitation of our tooling was the inability to profile unmanaged code. As we found out during our research, two out of four types of techniques we discovered utilized unmanaged code. Due to this limitation, we were unable to determine which unmanaged functions were called by these techniques. The profiling API for the .NET Framework provides minimal support for profiling unmanaged code, and instead instructs developers to write their own methods to achieve this. Given the time constraints for this research and the fact that this issue surfaced during analysis, we were unable to develop a solution for this. It is however possible to track the transition from managed to unmanaged code using the profiling API, which could partly fill this gap. In case P/Invoke is used, static analysis of the binary could also point out which functions are called during this transition.

A second limitation of our profiler was the inability to log the arguments to functions. While this is not necessary to determine the techniques that were used, it would allow for more in-depth analysis of the samples. For example, in case a D/Invoke-based technique is utilized, argument logging could point out which unmanaged functions are called. This is due to the fact that the D/Invoke technique has managed wrappers around the functions that find the addresses of unmanaged functions, which take the name of the DLL and hash of the function name as arguments. This information could in turn still enable us to determine which unmanaged functions were called. Therefore, this would also partly overcome the first limitation we mentioned, for this specific technique. Another benefit of argument logging would be the ability to dump the malicious payload to disk in some cases. When reflection is used, the arguments to `Reflection.Load` contain the assembly that the developer wants to load. Argument logging would enable us to log this argument, and therefore the malicious payload. While not a major limitation, the added benefits of fixing this limitation could lead to new insights and analysis opportunities.

Next, fileless techniques could have been missed due to the limited dataset used for this research. While the samples we analyzed belonged to the most popular post-exploitation frameworks for the .NET Framework, there might be samples in the wild that utilize currently unknown techniques. The application of our tooling on a larger scale could allow for these techniques to be identified, if they exist. This could be achieved through larger datasets from websites such as VirusTotal, in order to gain a larger amount of samples to analyze.

Lastly, while it did not occur during our analysis, it is possible for applications to detect the presence of the profiler and adjust their behaviour. As the environment variables that have to be set for the CLR to use to profiler can be accessed by any application, malware can access these variables as well. In case malware detects that the CLR is indeed instructed to use a profiler, it could avoid performing any malicious behaviour, which would therefore not be profiled. However, the check for the environment variables would be present in our call log, allowing us to detect this behaviour. Examining the logs could in turn show us how the sample detects the profiler, and make it possible for us to work around this check or remove it from the sample. Another option to avoid analysis would be for the malware sample to profile itself. A process can only be profiled by one profiler at a time, which would make us unable to analyze the sample in this case. A possible solution to tackle this limitation would be to modify the sample to either initialize our profiler, or not to initialize a profiler at all.

VIII. Future Work

As we discussed in the previous section, there are some limitations to this research that could be addressed in future research. We believe there are two potential research topics for future work, which build on top of the tooling developed for this research. These subjects both address the limitations of this research, as well as contribute valuable findings by themselves.

The first subject is directly linked to one of the limitations that was mentioned in the previous section, namely the size of the dataset. As the amount of samples that were analyzed for this research was limited, future work could be done to address this. Other fileless techniques might exist that we did not encounter in our dataset. This could be achieved by applying the tooling we developed to larger datasets, such as the VirusTotal academic database, or samples acquired through other sources. The contribution of this research to the academic world would be an even greater understanding of the currently available fileless malware techniques. Additionally, the resulting signatures could also lead to new ways to detect and prevent fileless malware.

Another subject which would be a valuable continuation of the work done in this research would be to utilize the methods we developed for malware detection. Instead of having a server component that simply stores to logs for later analysis like was done in this research, automated scanning could be added to the profiler itself. This would make it possible to scan for fileless malware threats and stop them as they occur. In order to achieve this, the signatures we developed would need to be applied on the calls as they are being made. This work could in addition also be extended with the automated extraction of malicious payloads, as we described in the previous section. This future research would be the next step in tackling the fileless malware issue.

IX. Conclusion

The goal of this research was to develop a method to tackle the currently ongoing fileless malware threat for the .NET Framework. We established that current state-of-the-art detection methods, such as AMSI, are insufficient at detecting the newest forms of fileless malware. Special attention was paid to a new fileless malware technique, which utilizes the Dynamic Language Runtime to dynamically interpret a malicious payload. In order to achieve our goal, we proposed a more generic approach to malware analysis and detection for the .NET Framework. Traditional anti-malware solutions target specific samples and strains by matching the hash of the sample against a list of known bad samples. Instead, our proposed method dynamically collects all the calls made by the application and detects fileless malware techniques in these calls.

We achieved this by developing a profiler to log the function calls of .NET Framework applications. This profiler is able to capture function calls for any .NET Framework language, which we verified by testing it on samples written in C#, F#, and Visual Basic. In turn, the profiler outputs a call tree containing all calls made by the application. This call tree captures the entire behaviour of the application, and allows us to compare the trees of different applications. As a result, this enables us to compare the trees of new samples to known malicious samples, in order to detect whether these new samples also contain similar malicious behaviour.

Next, after developing the required tooling, we were able to analyze the call trees to uncover the current fileless malware techniques for the .NET Framework, which was the second goal of this research. To achieve this, we analyzed the call trees produced by our tooling for the presence of fileless malware techniques for five different post-exploitation frameworks. After having identified the fileless malware techniques in the frameworks, we wrote YARA signatures for these techniques that can be applied to the output of our tooling. This then automatically pointed out whether a specific fileless malware technique was present in a profiled sample. As a result of this process, we were able to identify four distinct types of fileless malware techniques in our dataset:

- Reflection-based techniques
- P/Invoke-based techniques
- D/Invoke-based techniques
- Techniques utilizing an embedded interpreter

These techniques all utilize different .NET Framework API calls in order to achieve their fileless behaviour.

The third and final goal of this research was to compare these techniques in more detail, in order to understand the differences and similarities in their characteristics. As our signatures allow for the identification of these techniques and capture the calls that make these techniques possible, we were able to achieve this goal by comparing the signatures. This pointed out that both the reflection-based techniques and those utilizing an embedded interpreter use purely managed code, while the other two techniques include unmanaged, native code as well. Furthermore, while the D/Invoke-based technique works similarly to the P/Invoke-based technique, it avoids suspicious signatures by dynamically invoking native functions. This in turn makes this technique harder to be detected by intrusion detection systems, as there are no suspicious native API calls. The P/Invoke-based technique, on the other hand, utilizes standard .NET Framework features that allow native code to be called, and therefore also clearly shows this when statically analyzing the binary. This is similar to the reflection-based technique for managed code, as this technique utilizes the reflection API of the .NET Framework to load assemblies into memory, which AMSI is able to detect.

Additionally, as we are particularly interested in the technique based on the DLR, we compared the techniques utilizing an embedded interpreter in more detail, as one of these techniques utilized the DLR. This pointed out that the DLR technique indeed interprets a malicious payload, making it much harder to determine what is being executed compared to the other techniques. This is the case as the execution of every line of the payload is wrapped in interpreter related calls in our output, compared to the execution calls being in the same scope for the other techniques. Furthermore, the DLR technique made heavy use of the `Microsoft.Scripting` namespace, which contains the DLR related API calls. The presence of this namespace could in turn be used to detect this technique, and differential analysis could be used to filter out the calls belonging to this namespace and the interpreter.

There were however some limitations to our research, as the dataset we used to create signatures was limited. The result of this could be that there are fileless malware techniques for the .NET Framework which were not present in our dataset. Future research should be done to tackle this issue, and apply our tooling to a larger dataset. Additionally, there are two other improvements that could be made to the profiler to increase its effectiveness. The first of these improvements is the support for argument logging, which would make it possible to automatically capture the malicious payloads for some techniques and scan them. This improvement would also allow us to more effectively detect the usage of the D/Invoke-based techniques, as it uses arguments to specify which native function should be called. The other significant improvement that could be made is to the performance of the profiler. It is currently not possible to apply signatures in real-time while running the profiler. Real-time application of signatures would make it possible to incorporate the profiler into anti-malware solutions or intrusion detection systems.

We believe that our tooling is capable of becoming an outstanding tool at tackling the fileless malware threat for the .NET Framework. In this research, we applied our tooling to a limited dataset to show its capabilities and effectiveness. This showed that even with the current limitations, our tooling was able to detect threats that are capable of bypassing AMSI, which is one of the current state-of-the-art malware detection methods.

Acknowledgments

I would like to show my gratitude to my supervisors, Andrea, Anna, Erik and Robert for their continuous support during this project, as well as always being available to answer my questions and giving me incredibly valuable feedback to work with.

Next, I would like to thank my family for being supportive throughout my studies.

Finally, I would like to thank my friends for the fun moments we had throughout our studies, and hopefully more to come.

References

[1] GitHub - skbkontur/GroboTrace: Lightweight .NET performance profiler. Accessed: 2020-11-19.
[2] IronPython. https://web.archive.org/web/20200505112458/https://github.com/IronLanguages/ironpython2. Accessed: 2020-05-25.
[3] .NET programming languages — C#, F#, and Visual Basic. http://web.archive.org/web/20210426135437/https://dotnet.microsoft.com/languages. Accessed: 2021-04-26.
[4] PowerShell loves the Blue Team. https://web.archive.org/web/20200614142705/https://devblogs.microsoft.com/powershell/powershell-the-blue-team/. Accessed: 2020-06-11.
[5] PowerShellMafia/PowerSploit. https://web.archive.org/web/20200526155643/https://github.com/PowerShellMafia/PowerSploit. Accessed: 2020-05-26.
[6] AVG. AVG 2021 — FREE Antivirus & TuneUp for PC, Mac, Android. http://web.archive.org/web/20210610224301/https://www.avg.com/en-us/homepage. Accessed: 2021-06-13.
[7] Avira. Download Security Software for Windows, Mac, Android & iOS — Avira Antivirus. http://web.archive.org/web/20210611113922/https://www.avira.com/. Accessed: 2021-06-13.
[8] A. Baldin. Best practices for fighting the fileless threat. 2019(9):13–15.
[9] R. Barreto de Oliveira. The Boo Programming Language. https://web.archive.org/web/20090206045607/http://boo.codehaus.org/BooManifesto.pdf. Accessed: 2021-03-28.
[10] D. Bohannon and L. Holmes. Revoke-obfuscation: PowerShell obfuscation detection using science. Black Hat USA 2017.
[11] byt3bl33d3r. Byt3bl33d3r/OffensiveDLR. https://web.archive.org/web/20201205220116/https://github.com/byt3bl33d3r/OffensiveDLR/. Accessed: 2020-12-05.
[12] byt3bl33d3r. Byt3bl33d3r/SILENTTRINITY. https://web.archive.org/web/20200309082014/https://github.com/byt3bl33d3r/SILENTTRINITY. Accessed: 2020-05-25.
[13] R. Cobb. Cobbr/Covenant. https://web.archive.org/web/20201126113917/https://github.com/cobbr/Covenant/. Accessed: 2020-12-09.
[14] R. Cobb. Cobbr/SharpSploit. https://github.com/cobbr/SharpSploit/blob/manual-map/SharpSploit/Execution/PE.cs#L90. Accessed: 2021-04-15.
[15] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. 28(8):735–746. Conference Name: IEEE Transactions on Software Engineering.
[16] Cybereason Nocturnus. New Ursnif Variant Targets Japan Packed with New Features. https://web.archive.org/web/20201209154908/https://www.cybereason.com/blog/new-ursnif-variant-targets-japan-packed-with-new-features. Accessed: 2020-12-09.
[17] Cybersecurity & Infrastructure Security Agency. Emotet Malware — CISA. https://us-cert.cisa.gov/ncas/alerts/aa20-280a. Accessed: 2021-04-08.
[18] Cynet. Ransomware Attacks in Belgium - Analysis & Protection. https://web.archive.org/web/20201209160137/https://www.cynet.com/attack-techniques-hands-on/ransomware-attacks-in-belgium-analysis-protection/. Accessed: 2020-12-09.
[19] A. Dahan. Operation Cobalt Kitty: A large-scale APT in Asia carried out by the OceanLotus Group. https://web.archive.org/web/20201123135040/https://www.cybereason.com/blog/operation-cobalt-kitty-apt. Accessed: 2020-12-09.
[20] B. Delpy. GitHub - gentilkiwi/mimikatz: A little tool to play with Windows security. http://web.archive.org/web/20210407210015/https://github.com/gentilkiwi/mimikatz. Accessed: 2021-04-07.
[21] ESET. ESET NOD32 Antivirus. http://web.archive.org/web/20210613213546/https://www.eset.com/us/home/antivirus/. Accessed: 2021-06-13.
[22] J. Haight. Jaredhaight/PSAttack. https://web.archive.org/web/20200526161111/https://github.com/jaredhaight/PSAttack. Accessed: 2020-05-26.
[23] J. Heasman. Migrating to the .NET platform: An introduction. 2004(4):6–7.
[24] A. Kujawa. Under the Radar – The Future of Undetected Malware. http://web.archive.org/web/20210409195738/https://resources.malwarebytes.com/files/2018/12/Malwarebytes-Labs-Under-The-Radar-US.pdf.
[25] Z. Li, Q. A. Chen, C. Xiong, Y. Chen, T. Zhu, and H. Yang. Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for PowerShell Scripts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1831–1847. Association for Computing Machinery.
[26] Malwarebytes Labs. 2020 State of Malware Report. http://web.archive.org/web/20201122120501/https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report-1.pdf. Accessed: 2021-04-08.
[27] Microsoft. About Event Tracing - Win32 apps. http://web.archive.org/web/20201111190634/https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing. Accessed: 2021-04-16.
[28] Microsoft. Dynamic Language Runtime Overview. https://web.archive.org/web/20200424112116/https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-language-runtime-overview. Accessed: 2020-05-25.
[29] Microsoft. DynamicMethod Class | Microsoft Docs. https://web.archive.org/web/20210406192932/https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.dynamicmethod?view=net-5.0. Accessed: 2021-04-06.

[30] Microsoft. Overview of the .NET Framework. https://web.archive.org/web/20200517160525/https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview. Accessed: 2020-05-25.
[31] Microsoft. PE Format - Win32 apps | Microsoft Docs. http://web.archive.org/web/20210406133422/https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. Accessed: 2021-04-06.
[32] Microsoft. Platform Invoke (P/Invoke) | Microsoft Docs. http://web.archive.org/web/20210402114959/https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke. Accessed: 2021-04-02.
[33] Microsoft. Profiling overview. https://web.archive.org/web/20201215234000/https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview. Accessed: 2021-02-01.
[34] Microsoft. Reflection in .NET | Microsoft Docs. http://web.archive.org/web/20201112042249/https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection. Accessed: 2021-03-25.
[35] Microsoft. Script Tracing and Logging - PowerShell. http://web.archive.org/web/20201012204810/https://docs.microsoft.com/en-us/powershell/scripting/windows-powershell/wmf/whats-new/script-logging?view=powershell-7. Accessed: 2021-04-16.
[36] Microsoft. What is .NET Framework? A software development framework. https://web.archive.org/web/20200728143948/https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework. Accessed: 2020-07-28.
[37] Microsoft. What is PowerShell? - PowerShell. https://web.archive.org/web/20200526142151/https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7. Accessed: 2020-05-26.
[38] Microsoft. What's new in .NET Framework | Microsoft Docs. https://web.archive.org/web/20210402102452/https://docs.microsoft.com/en-us/dotnet/framework/whats-new/#common-language-runtime. Accessed: 2021-04-02.
[39] Microsoft. Antimalware Scan Interface (AMSI) - Win32 apps. https://web.archive.org/web/20200529130536/https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal, Apr. 2019. Accessed: 2020-05-29.
[40] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition.
[41] Ponemon Institute. The 2017 State of Endpoint Security Risk. http://web.archive.org/web/20201207003159/https://cdn2.hubspot.net/hubfs/468115/Campaigns/2017-Ponemon-Report/2017-ponemon-report-key-findings.pdf. Accessed: 2021-04-09.
[42] S. Pontiroli and R. Martinez. The rise of .NET and Powershell malware. https://web.archive.org/web/20200802233816/https://securelist.com/the-rise-of-net-and-powershell-malware/72417/. Accessed: 2020-08-03.
[43] S. M. Pontiroli and F. R. Martinez. THE TAO OF .NET AND POWERSHELL MALWARE ANALYSIS. page 26.
[44] A. Rousseau. Hijacking .NET to defend PowerShell. page 13.
[45] M. Salvati. Red Teamer's Cookbook: BYOI (Bring Your Own Interpreter). https://web.archive.org/web/20200321090539/https://www.blackhillsinfosec.com/red-teamers-cookbook-byoi-bring-your-own-interpreter/. Accessed: 2020-04-10.
[46] Sophos. Information on Attack Tool Detection. https://support.sophos.com/support/s/article/KB-000039774?language=en_US. Accessed: 2020-12-09.
[47] I. Stevenson. .NET core architecture. 14(5):24–27.
[48] V. Thakur. Malware analysis: Decoding Emotet, part 2. http://web.archive.org/web/20210408115359/https://blog.malwarebytes.com/threat-analysis/2018/06/malware-analysis-decoding-emotet-part-2/. Accessed: 2021-04-08.
[49] TheWover. Emulating Covert Operations - Dynamic Invocation (Avoiding PInvoke & API Hooks) – The Wover – Red Teaming, .NET, and random computing topics. http://web.archive.org/web/20210326154643/https://thewover.github.io/Dynamic-Invoke/. Accessed: 2021-03-26.
[50] TheWover. TheWover/DInvoke. https://github.com/TheWover/DInvoke/blob/ee256ba5a56cd45e813eb9eb007205090538d55d/DInvoke/DInvoke/DynamicInvoke/Generic.cs#L255. Accessed: 2021-04-06.
[51] VirusTotal. YARA - The pattern matching swiss knife for malware researchers. https://web.archive.org/web/20201210130549/https://virustotal.github.io/yara/. Accessed: 2020-12-10.